proceedings

3rd Symposium on Networked Systems Design & Implementation

San Jose, CA, USA May 8–10, 2006

Sponsored by
The USENIX Association

USENIX

in cooperation with ACM SIGCOMM & ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is \$40 for members and \$50 for nonmembers.

Outside the U.S.A. and Canada, please add \$20 per copy for postage (via air printed matter).

Past NSDI Symposia

2nd Symposium on Networked Systems Design & Implementation May 2–4, 2005, Boston, MA, USA

First Symposium on Networked Systems Design & Implementation March 29–31, 2004, San Francisco, CA, USA

Thanks to Our Sponsors











Thanks to Our Media Sponsors

ACM Queue GRIDtoday HPCwire ITtoolbox *Linux Journal* SNIA StorageNetworking.org
Sys Admin

© 2006 by The USENIX Association All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

USENIX Association in cooperation with ACM SIGCOMM & ACM SIGOPS

Proceedings of the 3rd Symposium on Networked Systems Design & Implementation (NSDI '06)

May 8–10, 2006 San Jose, CA, USA

Symposium Organizers

Program Chairs

Larry Peterson, *Princeton University* Timothy Roscoe, *Intel Research*

Matt Welsh, Harvard University

Program Committee

David Andersen, Carnegie Mellon University
John Byers, Boston University
Steve Gribble, University of Washington
Steve Hand, University of Cambridge
Mark Handley, University College London
John Hartman, University of Arizona
Rebecca Isaacs, Microsoft Research
Bryan Lyles, Telcordia
Adrian Perrig, Carnegie Mellon University
Jennifer Rexford, Princeton University
Dan Rubenstein, Columbia University
Emin Gün Sirer, Cornell University
Alex Snoeren, University of California, San Diego
Neil Spring, University of Maryland
Doug Tygar, University of California, Berkeley

Steering Committee

Thomas Anderson, University of Washington
Michael B. Jones, Microsoft Research
Greg Minshall
Robert Morris, Massachusetts Institute of Technology
Mike Schroeder, Microsoft Research
Amin Vahdat, University of California, San Diego
Ellie Young, USENIX Association

The USENIX Association Staff

External Reviewers

Tarek Abdelzaher Aditya Akella Scott Baker Rajesh Krishna Balan

Rajesh Krishna Balan Andy Bavier John Bethencourt Monica Brockmeyer David Brumley Justin Cappos Haowen Chan Constantine Dovrolis Shane Eisenman Nick Feamster Marc Fiuczynski Jason Franklin

Debin Gao Vijay Gopalakrishnan Huilong Huang Flavio Junqueira Dina Katabi Chip Killian Cynthia Kuo Seungjoon Lee Mark Luk Cristian Lumezanu

Jonathan M. McCune
David Marples
Margaret Martonosi
Tony McAuley
Vishal Misra
Ruggero Morselli
Richard Mortier
Steve Muir

Brendan Murphy Eric Nahum David Oppenheimer

Vivek Pai Bryan Parno Swapnil Patil Barath Raghavan Ramana Rao Kompella

Chris Sadler Runting Shi Robin Sommer

Lakshminarayanan Subramanian

Niraj Tolia Kevin Walsh Yong Wang Dan Wendlandt Dan Williams Bernard Wong Harlan Yu Pei Zhang Tao Zhang

NSDI '06: 3rd Symposium on Networked Systems Design & Implementation

May 8–10, 2006 San Jose, CA, USA

Index of Authors
Monday, May 8, 2006
Wide-Area Network Services I Session Chair: Steve Hand, University of Cambridge
Experience with an Object Reputation System for Peer-to-Peer Filesharing
Corona: A High Performance Publish-Subscribe System for the World Wide Web
Scale and Performance in the CoBlitz Large-File Distribution Service
Replication and Availability Session Chair: Steve Gribble, University of Washington
Efficient Replica Maintenance for Distributed Storage Systems
PRACTI Replication
Exploiting Availability Prediction in Distributed Systems
Tools Session Chair: Neil Spring, University of Maryland To Infinity and Beyond: Time-Warped Network Emulation
The Dark Oracle: Perspective-Aware Unused and Unreachable Address Discovery
Pip: Detecting the Unexpected in Distributed Systems

Tuesday, May 9, 2006

Wide-Area Network Services II Session Chair: Emin Gün Sirer, Cornell University
OASIS: Anycast for Any Service
OverCite: A Distributed, Cooperative CiteSeer
Colyseus: A Distributed Architecture for Online Multiplayer Games
End-System Design Session Chair: David Andersen, Carnegie Mellon University Na Kika: Secure Service Execution and Composition in an Open Edge-Side Computing Network
Raza, New York University Connection Conditioning: Architecture-Independent Support for Simple, Robust Servers
PCP: Efficient Endpoint Congestion Control
Measurement and Analysis Session Chair: John Byers, Boston University
Availability of Multi-Object Operations
Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems
Open Versus Closed: A Cautionary Tale

Tuesday, May 9, 2006 (continued)

Architectures and Abstractions (and Email) Session Chair: Alex Snoeren, University of California, San Diego
An Architecture for Internet Data Transfer
OCALA: An Architecture for Supporting Legacy Applications over Overlays
Distributed Quota Enforcement for Spam Control
RE: Reliable Email
Wednesday, May 10, 2006
Wireless and Sensor Networks Session Chair: Matt Welsh, Harvard University
PRESTO: Feedback-driven Data Management in Sensor Networks
Practical Data-Centric Storage
Geographic Routing Without Planarization
File and Storage Systems Session Chair: John Hartman, University of Arizona
Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks
Olive: Distributed Point-in-Time Branching Storage for Real Systems
Pastwatch: A Distributed Version Control System

100

 $(0)^{-1}$

Message from the Program Chairs

We are pleased to introduce these Proceedings of the 3rd Symposium on Networked Systems Design & Implementation. We are delighted to see NSDI continue to evolve into a first-rate venue for publishing results in networked systems, due almost entirely to the efforts of researchers writing and submitting very strong papers to the conference. The range of topics is nicely distributed across the traditional SOSP and SIGCOMM communities, and the quality of the work—particularly the commitment to building and evaluating working systems—continues to impress.

This year we received 110 papers, of which we selected 28 to appear in these Proceedings. Papers were subjected to two rounds of reviews by both the program committee and outside experts. During the first round, each submission received four reviews, which we used to select roughly half the papers for further consideration in the second round. A total of 524 reviews were written. The program committee met in Princeton, New Jersey, in January 2006 to discuss all the second-round papers. Each paper discussed at the PC meeting was read and personally reviewed by at least six members of the PC, leading to many lively and informed discussions. Finally, the accepted papers were individually shepherded by members of the program committee.

We are particularly indebted to the members of the program committee. They were diligent and conscientious in their reviews and deliberations, and without exception, a wonderful group of people to work with. Thank you! We also thank the external reviewers who provided valuable expertise. And, finally, we give special thanks to Ellie Young and Jane-Ellen Long of USENIX, who were somehow able to steer us through the myriad details of organizing a conference.

Larry Peterson, Princeton University Timothy Roscoe, Intel Research NSDI '06 Program Chairs

Index of Authors

Aguilera, Marcos K.	367	Karp, Brad	297	Raza, Sajid	169
Andersen, David G.	253	Killian, Charles	115	Reynolds, Patrick	115
Anderson, Thomas	197	Kravetz, Adam	169	Rosenblum, Mendel	353
Bailey, Michael	101	Krishnamurthy, Arvin	nd 197	Schroeder, Bianca	239
Balakrishnan, Hari	281	Kubiatowicz, John	45	Seshan, Srinivasan	155, 225
Belaramani, Nalini	59	Kubota, Ayumu	267	Shah, Mehul A.	115
Bharambe, Ashwin	155	Lakshminarayanan, F	Karthik	Shenker, Scott	281, 325
Chen, Benjie	381		129, 267	Shenoy, Prashant	311
Chun, Byung-Gon	45	Leong, Ben	339	Sirer, Emin Gün	1, 15
Collins, Andrew	197	Li, Jinyang	143	Sit, Emil	45
Cooke, Evan	101	Li, Ming	311	Snoeren, Alex C.	87
Councill, Isaac G.	143	Lichtman, Guy	169	Spence, Susan	367
Dabek, Frank	45	Liskov, Barbara	339	Stoica, Ion	267
Dahlin, Mike	59	Mazières, David	129, 297	Stribling, Jeremy	143
Ee, Cheng Tien	325	McNett, Marvin	87	Tolia, Niraj	253
Elliston, Amos	169	Michalakis, Nikolaos	169	Vahdat, Amin	87, 115
Freedman, Michael J.	129, 297	Mickens, James W.	73	Veitch, Alistair	367
Ganesan, Deepak	311	Miller, Jonathan	169	Venkataramani, Arun	59
Gao, Lei	59	Mogul, Jeffrey C.	115	Voelker, Geoffrey M.	87
Garfinkel, Tal	353	Morris, Robert 45,	, 143, 339, 381	Walfish, Michael	281
Garriss, Scott	297	Mortier, Richard	101	Walsh, Kevin	1
Gibbons, Phillip B.	211, 225	Nath, Suman	211, 225	Weatherspoon, Hakim	45
Grimm, Robert	169	Nayate, Amol	59	Wehrle, Klaus	267
Gupta, Diwaker	87	Noble, Brian D.	73	Wiener, Janet L.	115
Haeberlen, Andreas	45	Pai, Vivek S.	29, 183	Wierman, Adam	239
Harchol-Balter, Mor	239	Pang, Jeffrey	155	Yalagandula, Praveen	59
Jahanian, Farnam	101	Park, KyoungSoo	29, 183	Yip, Alexander	381
Joseph, Dilip	267	Patil, Swapnil	253	Yocum, Kenneth	87
Kaashoek, M. Frans	45, 143	Peterson, Ryan	15	Yu, Haifeng	211, 225, 297
Kaminsky, Michael	253, 297	Pfaff, Ben	353	Zahorjan, John	197
Kannan, Jayanth	267	Ramasubramanian, V	enugopalan 15	Zamfirescu, J.D.	281
Karger, David	281	Ratnasamy, Sylvia	325	Zheng, Jiandan	59

Experience with an Object Reputation System for Peer-to-Peer Filesharing

Kevin Walsh

Emin Gün Sirer

Cornell University {kwalsh,egs}@cs.cornell.edu

Abstract

In this paper, we describe Credence, a decentralized object reputation and ranking system for large-scale peerto-peer filesharing networks. Credence counteracts pollution in these networks by allowing honest peers to assess the authenticity of online content through secure tabulation and management of endorsements from other peers. Our system enables peers to learn relationships even in the absence of direct observations or interactions through a novel, flow-based trust computation to discover trustworthy peers. We have deployed Credence as an overlay on top of the Gnutella filesharing network, with more than 10,000 downloads of our client software to date. We describe the system design, our experience with its deployment, and results from a long-term study of the trust network built by users. Data from the live deployment shows that Credence's flow-based trust computation enables users to avoid undesirable content. Honest Credence clients can identify three quarters of the decoys encountered when querying the Gnutella network.

1 Introduction

Establishing trust is a fundamental problem in distributed systems. Peer-to-peer systems, in which service functionality is distributed across clients, eliminate the centralized components that have traditionally functioned as de facto trust brokers, and consequently exacerbate trustrelated problems. When peers lack meaningful measures on which to base trust decisions, they end up receiving services from untrustworthy peers, with effects that can range from wasted resources on mislabeled content to security compromises due to trojans. These problems are particularly evident in current peer-to-peer filesharing networks, which are rife with corrupt and mislabeled content [3]. Such content can waste network and client resources, lead users to download content they do not want, and aid the spread of viruses and other malware. Recent research confirms the vulnerability of deployed filesharing networks to corrupt and mislabeled content, and indicates that much of this pollution can be attributed to deliberate attacks [14].

The underlying problem facing clients of peer-to-peer filesharing systems is that they must assess the trustworthiness and intent of peers about which little is known. A simple approach is to use past experience with a peer to determine that peer's trustworthiness. But when client interactions are brief and span a large set of peers that changes dynamically, the opportunity to reuse information from past first-hand experience is limited. Another conventional approach is to interpret a shared file as an endorsement for that file's contents, similar to the way hyperlinks are interpreted as implicit votes by search engine ranking algorithms. Our data indicates that sharing is not a reliable indicator of a user's judgment. Peer-topeer filesharing networks call for trustworthiness metrics that are robust, predictive, and invariant under changing network conditions and peer resource constraints.

In this paper, we describe Credence, a new distributed reputation mechanism for peer-to-peer filesharing networks that enables honest, participating peers to confidently determine object authenticity, the degree to which an object's data matches its advertised description. Credence allows clients to explicitly label files as authentic or polluted, and to compute reputation scores for peers based on a statistical measure of the reliability of the peer's past voting habits. Combined, these two techniques provide relevant and reliable data so that clients can make informed judgments of authenticity before downloading unknown content. Credence's mechanism for computing peer reputations is fully decentralized, is not affected by extraneous or transient properties of peers, and is robust even when peers collude to misrepresent the authenticity of files in the network.

In order to gauge the trustworthiness of a peer in the absence of direct interactions or observations, Credence incorporates a flow-based trust computation. Conceptually similar to PageRank-style algorithms for propagating reputation through links on the Web [18], Credence's algorithm differs in fundamental ways from previous approaches. Credence is completely decentralized, and does not assume consensus on a set of pre-chosen peers from which trust flows in the network. Instead, each Credence client propagates trust from itself outwards into the network, using local observations to compute reputations in its immediate neighborhood, and input gathered from its community to judge more distant peers.

We have built and deployed a fully functioning Credence client as an extension to the LimeWire [15] client for the Gnutella filesharing network. We have been actively probing and monitoring the status of the Credence network since its public release. In this paper, we present a long-term study of the emerging properties of the Cre-

dence network using data collected over a period of nine months. This data validates the underlying assumptions used in the design of Credence, confirms that Credence's trust mechanism allows users to discover and maintain useful trust relationships in the network, and provides new insights into the behavior of filesharing users.

This paper describes the goals and assumptions of Credence, details the design and implementation of our Gnutella-based overlay, and presents the results of our long-term study of the evolution and structure of the Credence network. We show that the collective actions of Credence users have produced a robust network of peer relationships that lets honest peers avoid a majority of the pollution they encounter during typical queries. More fundamentally, this work demonstrates that it is feasible to construct an effective, fully decentralized reputation system in a large-scale peer-to-peer network.

2 Approach

Participants in a peer-to-peer filesharing network cooperate by routing queries from a client interested in an object to a set of peers serving it. Each object consists of some opaque content and, to facilitate searching, a formatted object descriptor containing the object name, encoding, content hash, and other descriptive meta-data. Clients issue keyword queries to their peers in the network, receive matching object descriptors in response, and then may download selected objects from among the responses.

Pollution is a problem when clients cannot reliably distinguish between descriptors for authentic objects, malicious decoys, mislabeled content, and accidentally damaged files. In an effort to identify authentic descriptors, typical clients rank search results according to the advertised file quality or the relative popularity of the file among the search results received. Such measures are easily manipulated by simple adversaries, rendering them unreliable at best and deceptive at worst. Recent work has shown that listing search results randomly is substantially more reliable than these rankings [9].

In this paper, we focus on the problem of distinguishing authentic objects from polluted ones. Other types of attacks can, of course, disrupt networks, and clients must use additional techniques to guard against them. Since we abstract away the underlying peer-to-peer network, our work is applicable to any file sharing network in which a decentralized object reputation system is called for and no pre-existing trust relationships can be assumed.

2.1 Objectives

The design of Credence is guided by several goals that are necessary requirements for a successful peer-to-peer reputation mechanism:

- Relevance: The system must use only pertinent information when evaluating the authenticity of objects and the credibility of peers.
- Distribution and Decentralization: No participants should be trusted a priori, and no central computation should be required during online operation.
- Robustness: The system must be robust to attacks by large numbers of coordinated, malicious peers.
- Isolation: The decision to participate in the reputation system should be independent of decisions to participate in unrelated activities, such as sharing files, contributing bandwidth, or remaining online.
- Motivation: Users must have realistic incentives to participate honestly in the reputation system.

To meet these objectives, the basic operation of a Credence client is organized around three main activities. First, Credence users vote on objects based on their own judgment of the object's authenticity. Second, Credence clients collect votes to evaluate the authenticity of objects they are querying. And third, clients evaluate votes from their peers to determine the credibility of each peer from the client's own perspective. In the base case, peer weights are computed by examining correlations in the voting histories of a client and its peers. In the general case, pairwise weights are combined to yield a trustworthiness metric using a graph flow algorithm.

In Credence, clients express their judgments about the authenticity of files using explicit voting, and the reputation system avoids any reliance on implicit indicators of a client's judgment. This is in sharp contrast with past systems that rely on sharing as an implicit endorsement of a file. As we show later, honest users often share corrupt or malicious files, files that they themselves were fooled into downloading. Similarly, the decision not to share a file is typically independent of the file's authenticity.

A client collects votes in order to evaluate the authenticity of prospective downloads. Credence uses a decentralized algorithm for propagating votes from the peers that cast them to the clients that seek them. Our implementation uses the underlying filesharing primitives to perform vote routing and search services, and so does not require any centralized coordination or computation.

Since deployed networks contain many unreliable and malicious peers, a client needs robust methods for evaluating the credibility of its peers. By definition, a peer's credibility with respect to judging file authenticity depends only on the votes it casts. In the absence of global consensus on the correctness of past votes, each client must decide from its own perspective if a peer's past votes were useful. Specifically, a client can compute the degree to which each peer's judgments match its own past votes, and then preferentially rely on votes from like-minded peers. In cases where direct pairwise eval-

uation is impossible due to insufficient overlap in voting activity, Credence employs a novel flow-based computation which extends trust relationships transitively through known peers to more distant peers.

The dependence on a client's own voting record provides a natural incentive to participate by voting often and carefully, since the client otherwise will find itself relying on similarly careless peers. In contrast to previous systems that rely on characteristics of past interactions, such as network bandwidth, peers in Credence are judged only by the votes they have cast. This isolates the reputation system from other decisions a filesharing peer must make, such as which peers to interact with, or how to allocate resources. Negative votes play an important role in identifying honest peers since, even if user interests differ or there is little overlap in the authentic files seen by different users, we expect near universal agreement on negative votes for spam and decoys among honest users. We show later that this is borne out in practice successful spam attacks on Gnutella help shape the Credence network, and help honest peers identify each other.

2.2 Vote Semantics

Credence works most effectively when a large fraction of users essentially share a common evaluation function, and so tend to agree when voting on objects in common. Widely accepted semantics for positive and negative votes will increase the chances of locating likeminded peers to rely on when evaluating object authenticity. We chose to base our system on file authenticity, rather than more subjective issues of taste or quality, for precisely this reason. This decision is in contrast to recommender systems, which try to identify a small number of peers with similar taste from which to make recommendations about new content. Recommender systems face significant challenges when clients have widely divergent tastes, and the need for a peer-to-peer approach is unclear given the existence of successful centralized recommender systems. Credence thus focuses solely on determining whether a file matches its description.

2.3 Cryptographic Keys

Credence ensures the integrity of votes by equipping every client with a cryptographic key pair K that is used to sign votes. Signatures prevent attackers from modifying existing votes or manufacturing new ones on behalf of other peers. Credence limits Sybil attacks [8] by requiring each client to possess a certificate cert_K signed by a central authority that vouches for K's validity. Our initial implementation rate limited the generation of certificates by requiring clients to download a large file for each requested certificate. The current implementation requires each client to solve a cryptographic puzzle, similar to the scheme proposed in [2]. The certificate authority in the download-for-key approach is entirely offline, while in

the puzzle-for-key approach, it is online but contacted only once during initial installation. In either case, the certificate authority plays no role in the filesharing network itself, and could be distributed using well known distributed authentication schemes if desired [29]. In all cases, client keys are not bound to real-world identities, but instead use randomly generated key pairs without any identifying information. These keys provide anonymity comparable to the anonymous pseudonyms found in existing filesharing networks.

We are now in a position to specify the actual protocol that Credence clients implement. In Section 3, we will evaluate how the protocol behaves in a real network, and examine evidence supporting the above assumptions using our long-term study of the deployed system.

2.4 Voting on Objects

The underlying goal of Credence is to allow a user to judge the authenticity of search results, each consisting of a file content hash and meta data, including the file's name, size, and type. Each search result can be viewed as a claim about the file's attributes. For example, $\langle H: gettysburg \subseteq name, mp3=type, 128=bitrate \rangle$ makes the claim that the file with content hash H has the specified attributes, where the symbol \subseteq is used to indicate that gettysburg is one of possibly many valid names for the file.

Credence clients express their observations by issuing votes, with each vote naming a file content hash and making specific claims about the file's attributes or contents. Other peers use these votes to evaluate the claims made by search results, by comparing if the claims specified the vote to those found in the search result. We say that a vote *applies* to a search result, either negatively or positively, if it either refutes or supports the search result's claims. For instance, a vote specifying $\langle H: mp3 \notin type \rangle$ applies negatively to the example search result above, since the two are mutually incompatible. Note that a vote's application may differ somewhat from the voter's original *intention*, as for example a vote $\langle H: jpg \notin type \rangle$ most likely intended to say something negative but would not apply at all to the above search result.

Clients must agree on a common syntax and semantics for making statements about objects. The language must be simple enough that ordinary users can encode their observations as votes, expressive enough to account for different types of pollution, and the semantics must be faithful to the user's intentions when voting. The voting language described here, and implemented in the current version of Credence, was carefully chosen to balance these three often conflicting goals.

Formally, each vote is a signed tuple $\langle H: S, T \rangle_K$ containing a file content hash H, a statement S about the file, and a timestamp T, together with the client's key certifi-

cate cert_K . A statement is an attribute value pair, combined with set operators \subseteq , =, \notin and \supseteq . The attribute and value are arbitrary strings, though Credence clients currently recognize three globally defined attributes: name, type, and bitrate. For efficiency, we allow multiple statements to be concatenated using an implied logical conjunction and signed together as a single vote.

The statements in votes enable users to designate particular values as valid or invalid for a given attribute, as follows. The statement $v \subseteq a$ says that v is a valid value for attribute a, though a may take on other values as well. Some attributes, most notably a file's name, are naturally multi-valued in the sense that many different values may be appropriate and valid for given unique file content hash. This operator can be used to specify one of the possibly many names. In contrast, v = a says that v is a valid value for attribute a and all other values are invalid. Attributes, such as the fixed bitrate of an audio file, can take on only a single possible value, and this operator enables users to express that single value. A negative statement $v \notin a$ says that v is an invalid value for attribute a. This operator is used to refute specific file advertisements, such as a single misleading file name or type. Finally, $v \supseteq a$ says that a may not take on values other than v. This operator allows clients to make strong, broadly applicable negative statements, without committing positively to the specified value. Such statements are particularly useful for thwarting relabeling attacks, where malicious peers change a file's metadata such that existing negative votes, when evaluated in a new context, might inadvertently apply positively to the modified metadata. Overall, these four operators enable Credence voters to express a wide range of statements through an easy-to-use graphical interface.

The Credence user interface gives users the option of voting in various ways on files that are shared locally, were recently downloaded, or are about to be deleted. The user can vote thumbs-up, which generates statements for the file's name, type, and bitrate, using \subseteq and = where appropriate. The user can vote thumbs-down to indicate that the file metadata was misleading, and select one or more attributes to include in the vote. Negative statements are generated using ∉ for the file's name and type. Since the true bitrate, by contrast, can be computed directly from the file, the vote can include a much broader negative statement using the \supseteq operator for the bitrate. Credence also supports an unconditional thumbs-down vote that generates the statement $\langle H: \text{name} = \emptyset \rangle$, which appropriately refutes any search result, since all valid such results contain a non-null name. This unconditional vote is meant to be used when the file contains a virus or otherwise wholly inappropriate content, under the assumption that no Credence user wishes to download a virus even if it is correctly labeled as such.

The user interface and vote operators were carefully designed with the goal of faithfulness in mind. In particular, thumbs-down votes do not ever support the claims of a search result, even when the known correct bitrate is included. Thus, a user that downloads a file with an incorrect name and incorrect bitrate does not inadvertently vote up the same file with a different incorrect name but the correct bitrate. In contrast, positive votes can apply negatively if the file's attributes are altered to be incompatible, such as would happen if the file type were modified after a thumbs-up vote was generated.

Early versions of the Credence software allowed only for an unconditional thumbs-down vote, as above, or an unconditional thumbs-up vote. This thumbs-up vote is handled as a special case and applies positively to any search result with the specified file hash. Since much of the data collected in our traces comes from our initially deployed clients, the analysis in Section 3 considers only the binary, up/down voting logic. Non-legacy votes are simply translated to unconditional votes as necessary.

2.5 Collecting and Storing Votes

In Credence, a client evaluating an object's authenticity actively queries the network to find, collect, then aggregate a sample of relevant votes. We implemented vote collection using the existing query infrastructure by issuing a *vote-gather* query, specifying the hash of the file of interest, to the underlying Gnutella network. This reactive, pull-based dissemination of votes is motivated by the Zipf popularity distribution of objects, since any given vote is unlikely to be of interest to many users.

The query is routed by Gnutella to peers sharing votes for the object, who respond by sending their own matching votes and any matching votes they have seen recently. If a peer knows many votes for the given hash, it sends only those with the most weight (from its own perspective), both in order to bound the overall cost of vote collection, and to ensure that the most useful votes are disseminated further in the network. Sending these additional votes improves vote availability and overall dissemination, and incurs little marginal cost. Specifically, voters are not required to remain online, since their votes can still be propagated by other peers.

In order to be able to respond to vote-gather queries as they arrive from the network, each peer maintains a *vote database* from which matching votes can be drawn. For each file content hash, the database stores a row with a timestamp, the peer's own vote, if any, and a list of other votes encountered recently for the object. Note that votes are maintained in the database regardless of how the peer voted on the file, or if the other votes agree with its own. As older entries expire, the database is constantly replenished from the peer's own voting activity and from votes the peer receives after issuing its own vote-gather

queries. The peer can further augment its database using a straightforward gossip exchange with its peers. The resulting database size is proportional to a peer's gossip rate and frequency of voting and estimation, and independent of the number of files in the network.

2.6 Weighing Votes

After collecting a set of votes for an object, the client verifies the signature and key certificate on each of the votes, then aggregates the set into a single reputation estimate to present to the user. Simply tabulating the available votes using unweighted averaging would be prone to manipulation, as attackers could simply flood the network with votes. Instead, each Credence client computes a trust metric for each vote, and uses weighted averaging to compute an estimate of the object's overall reputation. The resulting score is interpreted as a personalized estimate of the authenticity of the object, and can be used to make a more informed decision to accept (and fetch) or reject the object. In cases where no votes could be found, the user must resort to ad hoc estimates of authenticity, as used in past systems. Such cases are unavoidable during the initial deployment of any reputation system that does not rely on prior trust relationships.

The first step in aggregating votes is to evaluate how the claims in each vote apply to the relevant search result. Votes that apply positively are given an initial value of +1, and those that apply negatively -1. From the perspective of a client evaluating a set of votes, however, the usefulness of a particular vote depends on the relationship between the client and the peer that cast the vote, and so each client weighs the initial vote values according to the strength and bias of this relationship. Intuitively, peers that tend to vote identically (or inversely) on objects should develop strong positive (or negative) weights for each other's votes over time, while a client should disregard votes from peers that, from its perspective, appear to vote randomly.

2.7 Computing Correlations

Statistical correlation precisely captures this notion by comparing the shared voting history of each pair of peers. A Credence client determines, for each of its peers, a correlation coefficient θ to use as a weight during vote aggregation, based on the files voted on in common between the client and the peer. Conceptually, θ is calculated by examining the instances when both peers make statements about some file, and taking into account whether the statements have a positive or negative intention.

Normally, a single vote applies either positively, negatively or not at all, depending on the search result in question. The correlation computation takes place without reference to any particular file or search result, however, and so uses the original intent of each vote to directly compare the voting history of the client and peer. Specifically, we say that a pair of votes *conflict* if there is a search result to which the votes apply oppositely. We say the votes *agree* if they do not conflict and there is a search result that both support or both refute.

The coefficient θ is computed by examining all pairs of votes between two peers A and B that either conflict or agree with each other. Let a (respectively b) be the fraction of such votes from peer A (respectively B) with positive intention, and let p be the fraction of such pairs that agree with both votes having positive intention. Then $\theta = (p-ab)/\sqrt{a(1-a)b(1-b)}$ is the *coefficient of correlation*, taking on values in the range [-1,1]. This computation represents a standard technique for computing correlations on binary data. Positive values indicate agreement between peers, negative values indicate disagreement, and small $|\theta|$ indicates the absence of any significant relationship between the two voting histories.

Client A normally uses weight $r_{AB} = \theta$ for the votes cast by a peer B. When peers lack sufficient voting history to establish a robust estimate of θ , or when the correlation value itself is statistically insignificant, the client sets $r_{AB} = 0$ and so disregards votes from the peer. For peers whose votes are all negative or all positive, θ is usually undefined even if the peers are mostly or completely in agreement. Such cases may be common for clients newly joining the network, and so Credence uses a heuristic to allow such clients to quickly begin establishing tentative relationships. When θ is undefined, the software resorts to a simple vote agreement counting metric in the range $0 \le |r_{AB}| \le 0.75$. This range was chosen to be below the majority of existing correlation values, and reflects the decreased confidence in such heuristics as compared to the correlation computation. Clients may disable this heuristic, and it may be removed altogether if users can be convinced of the importance of voting both positively and negatively early on.

A client can only compute accurate and strong peer correlations if it has itself cast a sufficient number of both positive and negative votes. This restriction provides a strong incentive for users to participate in voting, since users that do not vote will find the quality of the estimates they compute noticeably degraded. A user can still benefit from Credence by voting honestly but privately, suppressing the sharing and dissemination of their votes to other users.

Credence clients use the information stored in their vote databases to periodically compute correlations for known peers. For each peer in the vote database, the client determines the set of objects for which it knows both the peer's vote and its own, derives from this set a peer correlation value, and caches any strong correlations found in a *correlation table*. The correlation table is consulted when weighing votes during the evaluation

of an object's authenticity, and for selecting votes to send in response to vote-gather queries from other clients.

2.8 Flow-based Peer Reputation

Computing correlations directly from the local vote database works well for peers that vote on overlapping sets of objects, and are thus well represented in the local vote database. But pairwise correlations cannot robustly evaluate the relationship between a client and peers having only a few interests in common with the client. We overcome this limitation by allowing clients to leverage the correlations discovered by their peers, effectively expanding their horizon along paths of correlated peers. Credence incorporates a notion of *transitive correlation* which enables strong correlations between a client and a nearby peer, and again between this peer and a more distant peer, to be combined into an estimate of the relationship between the client and the distant peer.

Transitive correlations are computed by building and maintaining a local model of the pairwise trust relationships between peers in the network, then periodically executing a flow-based algorithm on the resulting trust graph. Nodes in the trust graph represent peers in the network, and a weighted edge between nodes represents one peer's correlation estimate for another. Initially, a client populates the trust graph using locally computed correlations from its local vote database. The remainder of the graph is built using a gossip protocol, where each client randomly selects peers in the network and exchanges locally computed correlation coefficients. The selection of these gossip partners is biased towards peers with known positive correlations to preferentially expand the most useful parts of the graph.

Intuitively, votes from peers distantly connected in the graph can used to approximate the votes of peers more closely connected, by emulating the weighted voting computation at each step along the path. Performing a potentially large graph computation in this way during every search result evaluation is likely too expensive. We can approximate this computation by multiplying the weights along paths with strong weights in the graph, and so precompute an approximate effective weight to be used to weight the votes from each distantly connected peer. As a simplification and optimization in our implementation, each client periodically computes only a single maximum weight path to every other peer in its local graph, where path weight is the product of weights along edges. This computation is constrained to use paths where negative weights appear only on the last edge in the path, since a client cannot trust a negatively correlated peer to provide useful judgments about correlations to more distant peers. The resulting transitive correlations are cached for later use in weighting votes when a local correlation is not available.

Credence proposes two strategies to protect the reliability of its local trust graph against peers that lie about correlations when exchanging information. First, because only locally computed correlations are exchanged, a client can choose to audit the computation by requesting some or all of the inputs from its peer. Recall that inputs to the correlation computation are votes from peers, signed to maintain integrity. Second, in practice, the trust graph contains significant amounts of redundant information in the form of cycles and densely connected cliques. Auditing the graph itself can help the client identify misbehavior in the form of inconsistent information, and can also help guide decisions of which peers to audit directly. Auditing is not currently implemented in the deployed version of the Credence software.

The remainder of this paper presents our analysis of data gathered from the deployed Credence network.

3 Evaluation

Credence is the first peer-to-peer reputation system to be deployed widely on a live network, with over 10,000 downloads of our software since its initial public release in March, 2005. In this section we present an analysis of data collected in a long-term study of the deployed Credence network. The data presented here gives a unique view into the individual and collective behavior of file-sharing peers, and demonstrates the feasibility of a fully distributed reputation system in real settings.

We collected data on Credence clients using a continuously running crawler and have compiled more than 200 daily snapshots of the structure of the network over a span of nine months. Each snapshot contains the cumulative set of votes discovered by the crawler. Clients are identified in the data set only by their randomly chosen public key. Since a vote identifies the file to which it applies only by the file's hash, our vote data set does not contain the corresponding names of the files being voted on. However, over a span of six months a second crawler collected the names and hashes of files publicly shared by Credence clients, enabling us to contrast the sharing and voting habits of users. In order to compile a consistent view of the network for analysis, both crawlers ran simultaneously, and all analysis was performed off-line after data collection was finished. Cumulatively, the data contains over 39,000 votes cast and 84,000 files publicly shared by over 1200 Credence clients.

Our dataset likely comprises only a portion of the Credence user population, since peers that join and leave the network rapidly may be missed by our crawler. In general, Credence clients do not make complete information about their trust computations available, such as the list of known transitive relationships. In the analysis below, we simulate the perspective of clients in the network by recomputing the pairwise and transitive correlations each

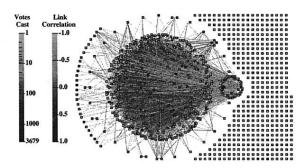


Figure 1: A global view of the Credence reputation graph with nodes and links shaded to reflect the number of votes cast by a peer and the correlations between peers. The large set of nodes in the center tend to correlate positively with each other, while clusters around the periphery are internally coordinated but conflict with the rest of the network.

peer normally computes, under the assumption that our vote set is representative of the complete set of votes in the network. This has the additional advantage of allowing us to evaluate different parameters and thresholds for the pairwise and transitive correlation computations.

In the next section we present a high level view of the Credence reputation graph as it exists at the end of our data collection, and in subsequent sections we evaluate the effectiveness of the Credence approach, examine assumptions behind the Credence design, and discuss the underlying factors driving the evolution of the network.

3.1 Graph Structure

Central to the Credence protocol is the ability to discover relationships between peers, so we begin with a global view of the trust relationship graph, derived from the cumulative set of votes collected by our crawler. Figure 1 presents the correlation values between any pair of peers with overlapping vote histories, formatted to reflect the clustering due to positive and negative correlations.

The most striking feature of the network is that, aside from the completely isolated nodes at the right, the graph is completely connected and has a very dense link structure. On average, each connected node is directly correlated with 27 other peers in the network. When combined with Credence's flow-based algorithm, this enables peers to derive reputations for a significant portion of the entire network. The isolated clients have no correlations, a result mainly of their very low voting activity. At the end of our study, isolated clients had cast on average fewer than 5 votes, compared with 82 votes on average for the connected nodes. The isolated clients are typically new clients, and make their way into the main cluster as they produce a more substantial voting record.

Among the active, connected clients, several vote completely oppositely as their peers, resulting in a negative correlation value for every incident edge. These peers are placed at the left of the figure. The remain-

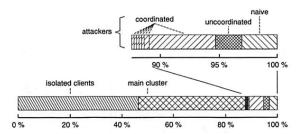


Figure 2: Classification of Credence users. Isolated client have no correlations to any other clients. Clients outside the main cluster are further classified according to type: naïve attacker for those that achieve only negative correlations; uncoordinated attacker for those that achieve some positive correlations but appear to be acting independently; and coordinated attacker for those that appear to have coordinated voting patterns different from those in the main cluster.

ing peers have a mix of positive and negative correlations with their peers. The large central component contains nodes with mainly positive correlations among each other. The smaller clusters of nodes that can be seen around the periphery of the central cluster have largely positive correlations internally, but mainly negative correlations with the rest of the network. Note that we have presented a global view of the graph structure, and do not show how any particular client would view the network, since each client uses its own votes to independently decide which nodes it considers inside its own cluster (positively correlated) and outside its cluster (negatively correlated). We show in the next sections that users in the main cluster can make such classifications accurately.

Coordination and Disagreements

Further examination of the reputation graph produced by Credence voters reveals the overall level of coordination and disagreement in the reputation system. If many users share a common notion of authenticity and pollution, then clients will more easily find correlated peers in the network. Figure 2 provides a classification of Credence users based on clustering observed in the global reputation graph. Some users vote so rarely or on such obscure files that they cannot derive any correlations, and are classified as *isolated*. A large majority of the remaining users are members of the single central cluster, and tend to agree on the authenticity of most files.

Approximately 3% of users are classified as *naïve attackers*, since they are easily identified as voting in direct contradiction to all other connected nodes. We also find an additional 10% of nodes that vote more often in contradiction to the overall network than they vote in agreement. More than half of these belong to a single large cluster of *coordinated attackers*, which can be seen to the right of the central cluster in Figure 1. The remaining nodes either participate in smaller coordinated attacks, or act as *independent attackers*.

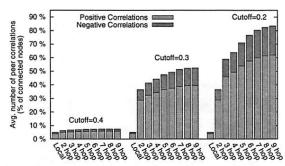


Figure 3: Number of local and transitive correlations computable under varying correlation strength threshold.

The Credence network has no authoritative source of content or a priori trusted peers. In particular, although we have labeled nodes outside the main cluster as attackers, it is not possible from our data to ascribe malicious intent to these nodes. They may simply have a different concept of file authenticity and pollution than the majority of nodes, or may not understand the voting process. From the perspective of the nodes in the central cluster, however, peer intent is irrelevant and all outside nodes can be fairly labeled as attackers. In Credence, however, such attackers are not necessarily damaging to individuals or the network as a whole. The votes from naïve attackers, for instance, are simply inverted by all other clients in the system, and so actually provide a tangible benefit to the system. Overall, the high level of agreement indicates that file authenticity is a fairly universal concept among filesharing users, justifying Credence's reliance on voting and correlation as a method of identifying authentic files and credible peers.

Local and Transitive Relationships

The set of peer correlations computed by a client plays a significant role in Credence, since it defines the set of votes that are normally used by the client when evaluating objects in the network. In this section, we show that clients participating in Credence's reputation system are able to identify both positively and negatively correlated peers in the network, and so can take advantage of a large fraction of the votes in the network.

Credence clients use direct, pairwise correlations to peers when possible, and use transitive correlations to propagate trust through these correlations to more distant peers in the network. Figure 3 shows the impact of both pairwise and transitive correlation computations on a client's view of the network, under varying strength criteria. We can see that the number of correlations computable directly from local information is fairly small on average, but that, depending on the choice of threshold value, a much larger set of correlations can be computed by clients using transitive information. The greater number of positive correlations found is due mainly to the overall trend toward agreement and cooperation ob-

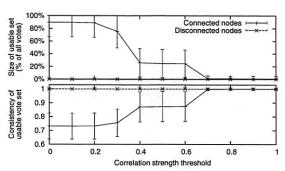


Figure 4: Characterization of votes usable by a client using various correlation strength thresholds.

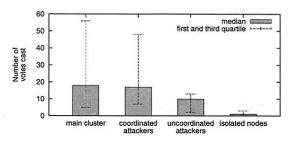


Figure 5: Number of votes cast according to client type. The low voting rate of isolated clients can be attributed to users that are either inactive, or have just recently joined the network.

served in the network, and in part to a bias against negative values in the transitive correlation computation.

Setting the correlation threshold allows a client to make an important tradeoff: a larger number of peer correlations could allow a client to take advantage of more votes from the network, but can also decrease the quality of estimates by including votes from weakly correlated and frequently inconsistent peers. Figure 4 illustrates this tradeoff by comparing the size of the set of usable votes for a given correlation threshold, and the consistency of this set of votes. To measure consistency, we compute the number of pairs of votes in agreement divided by the number of pairs in agreement or conflict. As a client increases its correlation threshold, it will have fewer peer correlations available, prompting both a decrease in the number of votes that can be used, and an increase in the consistency of the usable vote set.

Assimilation of New Clients

Clients newly joining the Credence network must be able to quickly discover peers with which they are correlated, so that the accuracy of their authenticity calculations can quickly increase. Our previous work, based on simulations, showed that transitive correlations play an important role in allowing new clients to quickly join the network [24]. After an initial period to establish a local voting record and a few pairwise correlations, the flow-based computation can immediately take advantage of the correlation results computed by already established

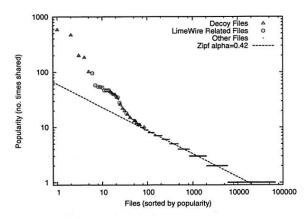


Figure 6: File popularity by number of times shared is approximately Zipf in the Credence network, but the most popular files are nearly all either decoys or related to LimeWire, and are noticeably overrepresented in the distribution.

peers. Figure 5 shows the number of votes cast by each type of client at the end of our study, and highlights the small number of votes necessary to become connected, and the relative inactivity of clients that are not yet connected. Looking at the data over time, 70% of connected clients discover their first peer correlation after casting fewer than 18 votes, which is the median for honest clients in the figure above.

3.2 Files in Credence

In this section we examine the overall distribution of files in the network, estimate the extent of pollution in our data set and its impact on clients, and examine how this pollution helps shape the overall structure of the Credence trust network. In Section 3.3, we show that clients are able to avoid this pollution using Credence's object authenticity rankings.

We collected lists of shared files from a set of 681 Credence clients. These users advertise a total of 84,838 files, of which 67,794 are unique, roughly following a Zipf popularity distribution as shown in Figure 6.

Decoys and Artifacts

The most frequently shared files are noticeably overrepresented, and are shared an order of magnitude more often than would be predicted by a strict Zipf distribution. Here we show that this deviation is due almost entirely to the effect of decoy attacks, demonstrating the substantial impact of pollution on the overall characteristics of the filesharing network. The remaining discrepancy can be explained by several files that appear to be widely, but inadvertently, shared by Credence users.

We manually examined all files shared under at least nine distinct names, and found all to be clear examples of decoy attacks – typically, movies or pictures containing advertisements (not surprisingly, the advertisements were often misleading; a movie containing an ad for

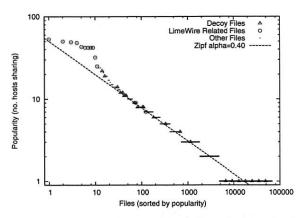


Figure 7: File popularity by number of hosts sharing a file is Zipf over the entire range, with the exception of certain files included in our software distribution.

a "free" iPod giveaway is highly prevalent, in part because a set of nodes on the Gnutella network sends it in response to every query and leads inattentive users to download it). Some of the other highly prevalent files are artifacts of the LimeWire and Credence software distributions, such as icons, source files, and software updates. Figure 6 labels all known decoy and LimeWire related files, discovered through manual examination of the 100 most frequently encountered files. These cases account for nearly all of the deviation from a strict Zipf popularity distribution.

The apparent popularity of many decoy files does not come from wide distribution in the network, but rather from a small number of hosts sharing the decoy files many times under different names. We can more accurately see the effects of decoy attacks on the network by measuring file popularity as the number of hosts sharing the file, shown in Figure 7, rather than as the number of replicas of the file. The figure shows two distinct types of decoy files. Those decoys that remain at the left of the distribution have spread to several peers in the network. The remaining decoys are shifted to the low end of the popularity distribution, meaning that they are shared by only a few peers. This shows that the apparently high popularity of some decoy files observed in Figure 6 is due to large-scale replication on a small number of clients, and provides clear evidence of malicious pollution in the underlying network.

Overall, our sharing data provides strong evidence that ongoing decoy attacks are targeting Credence clients, and that some of these decoy files are propagating successfully through the network. Other decoys rely on large scale replication by only a small number of clients, and these files do not propagate widely. We account for the difference in propagation in Section 3.3, showing that the larger decoy attacks are being suppressed by voting from honest Credence participants.

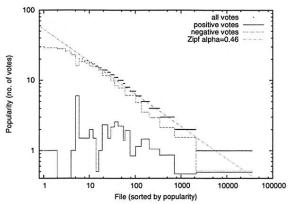


Figure 8: File popularity by number of votes received is Zipf. The most popular files, all decoys and artifacts, are slightly underrepresented. Solid and dashed lines show the number of positive and negative votes for each file, averaged within each popularity level, and indicate that positive votes are spread evenly across the entire range of file polarities, while negative votes cluster especially at the most popular files.

File Voting Popularity

Our voting data set comprises 39,761 votes cast on 35,690 unique files. The file most often voted on had 30 votes, while 33,530 files received only a single vote. The distribution of votes among files follows a Zipf popularity distribution, as shown in Figure 8, with the exception of the most popular files which are slightly underrepresented. Also shown is a breakdown of positive and negative voting frequency for each popularity level. This data shows that positive votes are spread across files fairly evenly, without regard to the popularity of the file. Negative votes, however, have a more skewed distribution, with many negative votes concentrated on a small fraction of all files.

These results indicate that many Credence users are encountering the same polluted files, consistent with a deliberate decoy attack. We observe in our data set that a few particular decoys are extremely common in search results, and these files are precisely those shown as the most popular in Figure 6, indicating that users are voting on those decoys that most affect them.

Voting is Independent of Sharing

A key design goal of Credence is to ensure that users can control the factors that influence their network reputation, and that users rely only on relevant data when judging the authenticity of files. Past systems have relied on sharing as proxy for users' explicit evaluations of files. We show in this section that not only are sharing and voting behavior largely independent of each other in our data set, but that they are often contradictory as well. This validates our reliance on explicit voting rather than implicit sharing indicators, and confirms that many users do not effectively monitor their sharing behavior.

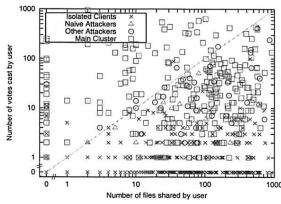


Figure 9: Credence user voting and sharing rate are largely independent, with users outside the main cluster found mainly along the x axis with no voting activity, and in the lower right of the graph with less voting activity than sharing activity.

Comparing the sharing and voting activity of individual users in Figure 9 shows that there is little correlation between the number of files voted on and the number of files shared, especially among the users in the large central cluster. Notably, nearly all Credence users outside the central cluster display much more sharing than voting behavior, whether the user is disconnected, or connected to the central cluster through purely or mostly negative correlations. This confirms that users who vote infrequently will tend to find themselves isolated in the reputation network, and provides incentive for honest users to participate in voting.

Voting Can Contradict Sharing

There is some overlap in voting and sharing activity of individual clients, but we find that a client's explicit votes often directly contradict the implicit indicators that sharing would have given. From our data, we can extract some 1754 instances spread over 123 users of a single user sharing and voting on the same file. A rational, honest user would only share files they vote positively on. This behavior accounts for roughly two thirds of the observed files. The remaining third of cases represent contradictory behavior, where a client has voted down a file they are also sharing. Interestingly, strictly rational behavior is observed on less than half (46%) of the users in the set. More surprising however is that nearly three quarters (72%) of the users display at least one case of inconsistent behavior, actively sharing files they explicitly voted against. This justifies our reliance on explicit voting rather than implicit sharing as an indicator of a user's judgment of file authenticity.

3.3 End-to-End Performance

As an end-to-end test of the effectiveness and utility of Credence, we examine how Credence performs when users execute typical queries in the Gnutella network, and how the Credence algorithms modify the relative ordering of the search results returned to the user. We used a load generator to repeatedly query the Gnutella network for typical keywords over a 24 hour period, and logged the search results returned. Query strings were generated from our earlier data set by extracting the four longest words from each file name shared by a Credence user. We selected terms in this way to mimic the interests of actual Credence users. This should not bias our query results, as sharing and voting behavior within Credence clients are essentially unrelated to each other, as our previous analysis has shown.

Queries that returned no search results were discarded. Each search result returned to our server consists of a file hash, a file name, various quality metrics, and a set of network addresses for the file. Using our voting data set, we matched the file hits returned to votes found in the data set generated by our crawler.

In all, Credence was able to provide some input to the search result ordering for 50.2% of the queries. Of the queries for which Credence could provide no input, the majority (79%) matched two or fewer files in the network. Of all the query replies received, more than a third were for files that had been voted on by Credence users. This result is quite encouraging, considering the very small number of Credence voters in comparison to the size of Gnutella network being queried.

Resistance to Decoy Attacks

In the above analysis, the distribution of query replies is skewed due to the popularity of certain files. Here, we examine the impact of decoy attacks on our query results and the ability of Credence clients to identify the decoy attacks encountered by our load generator. The specificity and scale of decoy attacks vary: some decoys are returned only for specific queries, while others are encountered for nearly all queries, and in both cases the number of responses naming the decoy can range from just one to several hundred. Thus, among the entire set of search results returned to our engine, only 12% contained unique file hashes, and only 1% of these are files for which Credence users have cast votes. In other words, although Credence users voted on only a small fraction of files encountered, less than 80 in all, these files represent more than a third of the total query replies. In previous sections we showed that Credence users tend to vote negatively more often on a small fraction of files. This suggests that many Credence users are likely being affected by a few very large decoy attacks, and are responding by voting negatively on them.

Using the list of decoy files identified by hand from our earlier analysis, we examine the impact of decoy attacks on filesharing search results, and characterize Credence's ability to identify such files when they are encountered. For this experiment, we selected individual

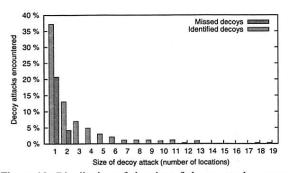


Figure 10: Distribution of the size of decoy attacks encountered, showing fraction of attacks successfully identified as pollution and those missed by Credence clients in the main cluster. Error bars are not visible, because these clients show nearly identical performance in identifying the decoys encountered.

clients in the central cluster to serve as vantage points, and computed how the clients would score each search result. In all cases where a decoy was encountered during a query, the clients were able to successfully identify 246 (75%) as pollution on average. The clients were not able to identify as pollution the remaining 82 (25%) decoy attacks, since no one appears to have voted on them. The deviation in results between clients was near zero, because the negative votes for the decoys come from peers that are weighted positively by honest clients.

Resistance to Collusion

Not all decoy attacks have equal impact on Credence users. In this section, we show that the decoys missed by Credence are those that have little impact on users in any case. Figure 10 shows the distribution of the sizes of each decoy result encountered, measured in terms of the number of locations purportedly offering the file. The data shows that any decoy offered by three or more peers is successfully identified. Credence misses several smaller decoy attacks, but even at these low levels still identifies nearly twice as many as it misses.

When Credence fails to identify some search results as polluted or authentic, it sorts these results according to their apparent popularity, just as non-Credence clients do. Larger attacks are more likely to rise to the top of such rankings, attracting the attention and negative votes of honest users, while the smaller attacks are more likely to be ignored by users. This tendency accounts for the lack of votes for such small decoy attacks.

We can conclude from these results that Credence users in the large central cluster of honest users are able to identify most decoy files as pollution by virtue of their own negative votes, or by the negative votes of other honest users, and that Credence is especially successful when attacks become sufficiently damaging to draw the attention of typical users.

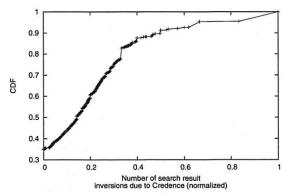


Figure 11: CDF of the normalized number of inversions in the search result rankings due to Credence. Credence considers legacy ordering at least 15% incorrect for half of the queries.

Ranking Performance

In addition to searching for and collecting votes for individual search results selected by the user, Credence is able to estimate file authenticity based on votes in the client's local vote database, and modify the sorting order of the results accordingly. Here we show that Credence's authenticity estimates have a significant impact on the presentation of typical search results. Specifically, we look at how the ordering in a non-Credence client, ranking files by the number of peers sharing the file, differs from the ordering in a Credence client, ranking files using known votes weighted by known peer correlations.

We use the number of inversions required to transform one sorted listing to the other, normalized to the range [0, 1], as a measure of difference. A score of 0 would indicate that Credence's result ordering coincides exactly with the non-Credence ordering, a score of 1 indicates that the Credence client orders the list exactly backwards compared to the legacy ranking, and a score of 0.5 would result if the legacy ranking were purely random with respect to Credence's ranking. Credence's default behavior of sorting according to the legacy ordering when no votes are available introduces a conservative bias towards zero in this difference metric. Figure 11 shows a CDF of the normalized number of inversions due to Credence, showing that the legacy ordering is no better than purely random, with respect to Credence's order, in about 10% of cases. We conclude that Credence can provide users with substantial credibility rankings in most cases.

3.4 Response to Attack

The design of Credence promotes subtle feedback loops and incentives that give rise to the resilience and overall dynamics shown in the previous sections. In our data set, we observe that polluted files that begin with an apparent high popularity, and a correspondingly high legacy ranking, are quickly discovered and voted down sufficiently often by honest users so as to put them at the bottom of the sorting order. An attacker's positive votes in this case

will serve only to induce additional negative votes from honest participants, and more importantly, reveals the attacking peer to not be credible. This effect can be seen as the nodes in main cluster of the network identify peers with consistently strong negative correlations.

A client's correlation values play a key role in its resistance to attack, and renders many attacks ineffective. A naïve attacker that votes consistently in opposition to honest clients will find itself cut off from the main cluster of honest nodes. At the other extreme, an attacker that votes randomly also becomes isolated, since it will generate no correlations with other peers. A rational attacker must carefully trade off honest votes on some files with dishonest votes on others, and so is required to leak a certain amount of correct information to the network.

The strongest attack we have explored is a whitewashing attack, in which the attacker first votes honestly on a large set of sacrificial files before voting dishonestly on a smaller target set. In our study, we find that such attackers may be included or excluded depending on each client's perspective. An individual client excludes the attacker if, from the clients own perspective, the information gained from the attacker's honest votes is outweighed by the damage from the dishonest votes. In other instances, clients find that the attacker's negative votes does so little damage and provides so much honest information as to make it worthwhile to include the attacker as a partially credible peer. Client perspectives vary, making it more difficult for attackers to select the sacrificial files on which to vote honestly. Further, multiple independent attacks of this form can easily cancel each other, and lead to an overall net gain of honest information in the system.

A more complete discussion of this and other potential attack scenarios can be found in a previous work [24], based on simulations of the Credence protocol.

3.5 Credence Overhead

Credence performs various operations in the background, and so requires some processing and network resources on client machines. A client representative of the most active existing users, having cast 250 votes and having learned over time of an additional ten thousand votes, can expect to receive approximately 100 bytes per second of additional background traffic due to Credence. This is in comparison to LimeWire's approximately 60 bytes per second of incoming background traffic. Outbound traffic depends strongly on the popularity of the client's votes, the client's reputation, and Gnutella connectivity. Ongoing background processing of additional Gnutella queries and Credence gossip requests in the same scenario demands less than 1% of a 1.7GHz processor, while a complete recomputation of all correlations can be completed in under three seconds.

4 Related Work

Deployed peer-to-peer systems are known to be vulnerable to many forms of malicious activity. A recent study [3] gives clear evidence of intentional pollution attacks in four large filesharing networks, and discusses the lack of reliable tools available for peers to avoid this pollution. Similarly, Ross [14] finds evidence of rampant pollution in the decentralized Kazaa network.

Distributed Peer Reputation: Past work on peer-topeer reputation focuses mainly on service differentiation, which refers to the ability of clients to make intelligent decisions about which peer among many to select for service. Typically, the goal of such systems is to discourage freeloaders by excluding them from the network, or optimize download performance by selecting high performing peers. Thus, past work relies mainly on peer reputation based on performance measures of peers, rather than object reputation as we propose.

Eigentrust [13] computes a single performance score for each peer, reflecting their past behavior in pairwise interactions. Although the protocol is distributed, it ultimately relies on a fixed set of trusted nodes at which it roots the computation of trust. Collusion can also disrupt straightforward eigenvalue computations, and techniques to make eigenvalue-based systems more resistant to collusion [28] rely heavily on centralized computation.

Other approaches [1, 25, 7, 4] enable each client to compute a personalized, rather than global, performance score for peers in the network, and also distinguish peer performance and peer credibility. A client considers a peer credible if the client and peer tend to agree on most performance observations made in the past. However, peers in such a system are unlikely to be found credible due simply to the wide variation in network perspectives, changing performance characteristics over time, locality-based peer selection, and the high degree of object replication in typical filesharing networks.

XRep [6] and X²Rep [5] extend the work in [4] by additionally computing object reputations based on weighted peer voting, with the weights based on past voting behavior of peers. These protocols require peers to be online during each object evaluation phase in order to compute and transmit their votes, and do not share the computed weights among peers. Information sharing and offline operation are critical features for a peer-to-peer reputation system due to the sparse workloads and session lengths observed in peer-to-peer networks [21].

Alternative approaches to the freeloader problem have been proposed without resort to peer reputation. Gupta, Judge, and Ammar propose an economic model [12] where peers earn credits for participation and pay credits to gain service, and explore the tradeoffs between reliability and overhead when accounting is distributed in the network. Micropayments can be used to induce cooperation (e.g. [23, 26, 17]). Fair exchange systems [10] provide similar properties and incentives, but without relying on currency. These schemes do not address content pollution, and so are complementary to our approach.

Credence does not address freeloading or service differentiation problems, but rather content pollution. In a broad study of denial-of-service vulnerabilities in peer-to-peer networks, Dumitriu et al. [9] discuss pollution based attacks and factors that make them successful in existing networks, and note the tendency of clients to inadvertently share corrupted files. The authors estimate the potential impact of a general class of peer-based reputations systems, and find them to be insufficient to counter pollution-based attacks. Using simulation and epidemiological models of the spread of pollution in file-sharing networks, Thommes and Coates [22] show that Credence's object-based approach can have a significant impact on network-wide pollution levels, even when only a fraction of participants use Credence.

Centralized Pollution Control: Problems similar to filesharing pollution have long been recognized in other domains, prior to the emergence of modern peer-topeer networks. For instance, recommender systems (e.g. [20]) aim to distinguish wanted and unwanted content in Usenet and in other domains, and online marketplaces often provide some form of reputation system so buyers can avoid untrustworthy sellers. Although Credence is not a recommender system, our distributed flowbased correlation computations resemble the centralized recommender algorithms in [27] for use in online marketplaces. Guha et al. [11] examine how both positive and negative evaluations might be propagated through a web of pairwise observations, and we share a similar model of information propagation in Credence. In general, however, past approaches to pollution-like problems rely heavily on centralized components and are not directly applicable to peer-to-peer networks.

BitTorrent has remained relatively free of pollution, partly due to human moderation of BitTorrent lists and tight binding of trackers to nodes [19]. Recent trends indicate that decentralized tracking and auto-generated torrent lists are opening BitTorrent up to pollution [16]. Credence techniques can be applicable in such contexts.

5 Conclusions

Credence is a new approach for combating the widespread presence of decoys, malware, and other malicious content in peer-to-peer filesharing systems. The system provides incentives for peers to participate honestly in voting, enables peers to compute object reputations that reflect their authenticity, and is robust to coordinated attacks. We have made a complete Credence

implementation, with source code, freely available. Data from a long-term study of the emerging properties of the deployed network suggests that Credence users are able to identify malicious filesharing activity and mitigate the impact of dishonest peers in the Credence reputation system.

The techniques we have developed in Credence are not specific to the Gnutella network in which they are currently deployed, but are applicable to a broader class of distributed systems. These systems are characterized by the need for users to make local trust decisions about networked objects and services, without a priori trust relationships imposed by a central authority. As critical network infrastructure services become more decentralized, conventional centralized trust decisions will necessarily become unsuitable. Such systems can benefit from the Credence approach.

Acknowledgments

This work was supported in part by NSF Career grant 0546568, and TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (CCF-0424422) and the following organizations: Cisco, ESCHER, HP, IBM, Intel, Microsoft, ORNL, Qualcomm, Pirelli, Sun and Symantec.

References

- S. Buchegger and J.-Y. L. Boudec. A Robust Reputation System for P2P and Mobile Ad-hoc Networks. In Workshop on the Economics of Peer-to-Peer Systems, Boston, MA, June 2004.
- [2] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In Symposium on Operating Systems Design and Implementation, Boston, MA, December 2002.
- [3] N. Christin, A. S. Weigend, and J. Chuang. Content Availability, Pollution and Poisoning in File Sharing Peer-to-Peer Networks. In ACM Conference on Electronic Commerce, Vancouver, Canada, June 2005.
- [4] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Choosing Reputable Servents in a P2P Network. In *International World Wide Web Conference*, Honolulu, HI, May 2002.
- [5] N. Curtis, R. Safavi-Naini, and W. Susilo. X²Rep: Enhanced Trust Semantics for the XRep Protocol. In *Applied Cryptography* and *Network Security*, Yellow Mountain, China, June 2004.
- [6] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks. In ACM Conference on Computers and Communications Security, Washington, DC, October 2002.
- [7] R. Dingledine, M. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, July 2000.
- [8] J. R. Douceur. The Sybil Attack. In International Workshop on Peer-to-Peer Systems, Cambridge, MA, March 2002.
- [9] D. Dumitriu, E. Knightly, A. Kuzmanovic, I. Stoica, and W. Zwaenepoel. Denial-of-Service Resilience in Peer-to-Peer File Sharing Systems. In ACM SIGMETRICS, Banff, Canada, June 2005.

- [10] P. Gauthier, B. Bershad, and S. D. Gribble. Dealing with Cheaters in Anonymous Peer-to-Peer Networks. Technical Report 04-01-03, University of Washington, January 2004. Computer Science and Engineering.
- [11] R. Guha, R. Kumar, P. Raghavan, and A. Tomkins. Propagation of Trust and Distrust. In *International World Wide Web Conference*, New York, NY, May 2004.
- [12] M. Gupta, P. Judge, and M. Ammar. A Reputation System for Peer-to-Peer Networks. In ACM Intl. Workshop on Network and Operating System Support for Digital Audio and Video, Monterey, CA, June 2003.
- [13] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigen-Trust Algorithm for Reputation Management in P2P Networks. In International World Wide Web Conference, Budapest, Hungary, May 2003.
- [14] J. Liang, R. Kumar, Y. Xi, and K. W. Ross. Pollution in P2P File Sharing Systems. In *IEEE INFOCOM*, Miami, FL, March 2005.
- [15] LimeWire. http://www.limewire.com/.
- [16] T. Mennecke. New Breed of Corrupt Torrent Infiltrates BitTorrent, September 2005. http://slyck.com/news.php?story=296.
- [17] MojoNation. http://www.mojonation.net/.
- [18] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998. Stanford Digital Libraries Working Paper.
- [19] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System. Technical Report PDS-2004-003, Delft University of Technology, April 2004.
- [20] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In ACM Conference on Computer Supported Cooperative Work, Chapel Hill, NC, October 1994.
- [21] S. Saroiu, K. P. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Multimedia Computing and Networking*, San Jose, CA, January 2002.
- [22] R. Thommes and M. Coates. Epidemiological Models of Peer-to-Peer Viruses and Pollution. Technical report, McGill University, June 2005. Department of Electrical and Computer Engineering.
- [23] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In Workshop on the Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.
- [24] K. Walsh and E. G. Sirer. Fighting Peer-to-Peer SPAM and Decoys with Object Reputation. In Workshop on the Economics of Peer-to-Peer Systems, Philadelphia, PA, August 2005.
- [25] L. Xiong and L. Liu. PeerTrust: Supporting Reputation-Based Trust in Peer-to-Peer Communities. IEEE Transactions on Knowledge and Data Engineering, Special Issue on Peer-to-Peer Based Data Management, 16(7), July 2004.
- [26] B. Yang and H. Garcia-Molina. PPay: Micropayments for Peer-to-Peer Systems. In ACM Conference on Computers and Communications Security, Washington, DC, October 2003.
- [27] G. Zacharia, A. Moukas, and P. Maes. Collaborative Reputation Mechanisms in Electronic Marketplaces. In *Hawaii International Conference on System Sciences*, Maui, HI, January 1999.
- [28] H. Zhang, A. Goel, R. Govindan, K. Mason, and B. V. Roy. Making Eigenvector-Based Reputation Systems Robust To Collusion. In Workshop on Algorithms and Models for the Web-Graph, Rome, Italy, October 2004.
- [29] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. ACM Transactions on Computer Systems, 20(4):329–368, 2002.

Corona: A High Performance Publish-Subscribe System for the World Wide Web

Venugopalan Ramasubramanian Ryan Peterson Emin Gün Sirer

Cornell University, Ithaca, NY 14853

{ramasv,ryanp,egs}@cs.cornell.edu

Abstract

Despite the abundance of frequently changing information, the Web lacks a publish-subscribe interface for delivering updates to clients. The use of naïve polling for detecting updates leads to poor performance and limited scalability as clients do not detect updates quickly and servers face high loads imposed by active polling. This paper describes a novel publish-subscribe system for the Web called Corona, which provides high performance and scalability through optimal resource allocation. Users register interest in Web pages through existing instant messaging services. Corona monitors the subscribed Web pages, detects updates efficiently by allocating polling load among cooperating peers, and disseminates updates quickly to users. Allocation of resources for polling is driven by a distributed optimization engine that achieves the best update performance without exceeding load limits on content servers. Large-scale simulations and measurements from PlanetLab deployment demonstrate that Corona achieves orders of magnitude improvement in update performance at a modest cost.

1 Introduction

Even though Web content changes rapidly, existing Web protocols do not provide a mechanism for automatically notifying users of updates. The growing popularity of frequently updated content, such as Weblogs, collaboratively authored Web pages (wikis), and news sites, motivates a *publish-subscribe* mechanism that can deliver updates to users quickly and efficiently. This need for asynchronous update notification has led to the emergence of *micronews syndication* tools based on naïve, repeated polling. The wide acceptance of such micronews syndication tools indicates that backwards compatibility with existing Web tools and protocols is critical for rapid adoption.

However, publish-subscribe through uncoordinated polling, similar to the current micronews syndication, suffers from poor performance and scalability. Subscribers do not receive updates quickly due to the fundamental limit posed by the polling period, and are tempted to poll at faster rates in order to detect updates quickly.

Consequently, content providers have to handle the high bandwidth load imposed by subscribers, each polling independently and repeatedly for the same content. Moreover, such a workload tends to be "sticky;" that is, users subscribed to popular content tend not to unsubscribe after their interest diminishes, causing a large amount of wasted bandwidth. Existing micronews syndication systems provide ad hoc, stop-gap measures to alleviate these problems. Content providers currently impose hard ratelimits based on IP addresses, which render the system inoperable for users sharing an IP address, or they provide hints for when not to poll, which are discretionary and imprecise. The fundamental problem is that an architecture based on naïve, uncoordinated polling leads to ineffective use of server bandwidth.

This paper describes a novel, decentralized system for detecting and disseminating Web page updates. Our system, called Corona, provides a high-performance update notification service for the Web without requiring any changes to the existing infrastructure, such as Web servers. Corona enables users to subscribe for updates to any existing Web page or micronews feed, and delivers updates asynchronously. The key contribution that enables such a general and backwards-compatible system is a distributed, peer-to-peer, cooperative resource management framework that can determine the optimal resources to devote to polling data sources in order to meet system-wide goals.

The central resource-performance tradeoff in a publish-subscribe system in which publishers serve content only when polled involves bandwidth versus update latency. Clearly, polling data sources more frequently will enable the system to detect and disseminate updates earlier. Yet polling every data source constantly would place a large burden on publishers, congest the network, and potentially run afoul of server-imposed polling limits that would ban the system from monitoring the micronews feed or Web page. The goal of Corona, then, is to maximize the effective benefit of the aggregate bandwidth available to the system while remaining within server-imposed bandwidth limits.

Corona resolves the fundamental tradeoff between bandwidth and update latency by expressing it formally as a mathematical optimization problem. The system then computes the optimal way to allocate bandwidth to monitored Web objects using a decentralized algorithm that works on top of a distributed peer-to-peer overlay. This allocation takes object popularity, update rate, content size, and internal system overhead stemming from accounting and dissemination of meta-information into account, and yields a polling schedule for different objects that will achieve global performance goals, subject to resource constraints. Corona can optimize the system for different performance goals and resource limits. In this paper, we examine two relevant goals: how to minimize update latency while ensuring that the average load on publishers is no more than what it would have been without Corona, and how to achieve a targeted update latency while minimizing bandwidth consumption. We also examine variants of these two main approaches where the load is more fairly balanced across objects.

The front-end client interface to Corona is through existing instant messaging (IM) services. Users subscribe for content by sending instant messages to a registered Corona IM handle, and receive update notifications asynchronously. Internally, Corona consists of a cloud of nodes that monitor the set of active feeds or Web pages called channels. The Corona resource allocation algorithm determines the number of nodes designated to monitor each channel. Cooperative polling ensures that the system can detect updates quickly while no single node exceeds server-designated limits on polling frequency. Each node dedicated to monitoring a channel has a copy of the latest version of the channel contents. A feed-specific difference engine determines whether detected changes are germane by filtering out superficial differences such as timestamps and advertisements, extracts the relevant portions that have changed, and distributes the delta-encoded changes to all internal nodes assigned to monitor the channel, which in turn distribute them to subscribed clients via IM.

We have implemented a prototype of Corona and deployed it on PlanetLab. Evaluation of this deployment shows that Corona achieves more than an order of magnitude improvement in update performance. In experiments parameterized by real RSS workload collected at Cornell [19] and spanning 60 PlanetLab nodes and involving 150,000 subscriptions for 7500 different channels, Corona clients see fresh updates in intervals of 45 seconds on average compared to legacy RSS clients, which see a mean update interval of 15 minutes. At all times during the experiment, Corona limits the total polling load on the content servers within the load imposed by the legacy RSS clients.

Overall, Corona is a new overlay-based publishsubscribe system for the Web that provides asynchronous notifications, fast update detection, and optimal bandwidth utilization. This paper makes three contributions: (i) it outlines the general design of a publish-subscribe system that does not require any changes to content sources, (ii) formalizes the tradeoffs as an optimization problem and presents a novel distributed numerical solution technique for determining the allocation of bandwidth that will achieve globally targeted goals while respecting resource limits, and (iii) presents results from extensive simulations and a live deployment that demonstrate that the system is practical.

The rest of the paper is organized as follows. The next section provides background on publish-subscribe systems and discusses other related work. Section 3 describes the architecture of Corona in detail. Implementation details are presented in Section 4 and experimental results based on simulations and deployment are described in Section 5. Finally, Section 6 summarizes our contributions and concludes.

2 Background and Related Work

Publish-subscribe systems have raised considerable interest in the research community over the years. In this section, we provide background on publish-subscribe based content distribution and summarize the current state of the art.

Publish-Subscribe Systems: The publish-subscribe paradigm consists of three components: publishers, who generate and feed the content into the system, subscribers, who specify content of their interest, and an infrastructure for matching subscriber interests with published content and delivering matched content to the subscribers. Based on the expressiveness of subscriber interests, pub-sub systems can be classified as topicbased or content-based. In topic-based systems, publishers and subscribers are connected together by predefined topics, called channels; content is published on well-advertised channels to which users subscribe to receive asynchronous updates. Content-based systems enable subscribers to express elaborate queries on the content and use sophisticated content filtering techniques to match subscriber interests with published content.

Prior research on pub-sub systems has primarily focused on the design and implementation of content filtering and event delivery mechanisms. Topic-based publish-subscribe systems have been built based on several decentralized mechanisms, such as group communication in Isis [13], shared object spaces in Linda [5], TSpace [36], and Java Spaces [16] and rendezvous points in TIBCO [35] and Herald [4]. Content-based publish-subscribe systems that use in-network content filtering and aggregation include SIENA [6], Gryphon [34], Elvin [32], and Astrolabe [37]. While the above publish-subscribe systems impose well-defined struc-

tures for the content, few systems have been proposed for semi-structured and unstructured content. YFilter [8], Quark [3], XTrie [7], and XTreeNet [11] are recent architectures for supporting complex content-based queries on semi-structured XML data. Conquer [21] and WebCQ [20] support unstructured Web content.

The fundamental drawback of the preceding publish-subscribe systems is their non-compatibility with the current Web architecture. They require substantial changes in the way publishers serve content, expect subscribers to learn sophisticated query languages, or propose to layout middle-boxes in the core of the Internet. On the other hand, Corona interoperates with the current pull-based Web architecture, requires no changes to legacy Web servers, and provides an easy-to-use IM based interface to the users. Optimal resource management in Corona aimed at bounding network load insulates Web servers from high load during flash-crowds.

Micronews Systems: Micronews feeds are short descriptions of frequently updated information, such as news stories and blog updates, in XML-based formats such as RSS [30] and Atom [1]. They are accessed via HTTP through URLs and supported by client applications and browser plug-ins called feed readers, which check the contents of micronews feeds periodically and automatically on the user's behalf and display the returned results. The micronews standards envision a publish-subscribe model of content dissemination and define XML tags such as cloud that tell clients how to receive asynchronous updates, as well as TTL, SkipHours, and SkipDays that inform clients when not to poll. Yet few content providers currently use the cloud tag to deliver asynchronous updates.

Recently, commercial services such as Bloglines, NewsGator, and Queoo have started disseminating micronews updates to users. Corona differs fundamentally from these commercial services, which use fragile centralized servers and relentless polling to detect updates. Corona is layered on a self-organizing overlay comprised of cooperative peers that share updates, judiciously determine the amount of bandwidth consumed by polling, and can provide strong bandwidth guarantees.

FeedTree [31] is a recently proposed system for disseminating micronews that also uses a structured overlay and shares updates between peers. FeedTree nodes perform cooperative update detection in order to reduce update dissemination latencies, and Corona shares the insight with FeedTree that cooperative polling can drastically reduce update latencies. FeedTree decides on the number of nodes to dedicate to polling each channel based on heuristics. Corona's key contribution is the use of informed tradeoffs to optimal resource management. This principled approach enables Corona to provide the best update performance for its users, while ensuring that

content servers are lightly loaded and do not get overwhelmed due to flash-crowds or sticky traffic.

CAM [26] and WIC [25] are two techniques for allocating bandwidth for polling data sources on a single node. Similar to Corona, they perform resource allocation using analytical models for the tradeoff and numerical techniques to find near-optimal allocations. However, these techniques are limited to a single node. Corona performs resource allocation in a decentralized, cooperative environment and targets globally optimal update performance.

Overlay Networks: Corona is layered on structured overlays and leverages the underlying structure to facilitate optimal resource management. Recent years have seen a large number of structured overlays that organize the network based on rings [33, 29, 39, 23], hyperdimensional cubes [28], butterfly structures [22], deBruijn graphs [17, 38], or skip-lists [14]. Corona is agnostic about the choice of the overlay and can be easily layered on any overlay with uniform node degree, including the ones listed here.

Corona's approach to a peer-to-peer resource management problem has a similar flavor to that of Beehive [27], a structured replication framework that resolves space-time tradeoffs optimizations in structured overlays. Corona differs fundamentally from Beehive in three ways. First, the Beehive problem domain is limited to object replication in systems where objects have homogeneous popularity, size, and update rate properties, whereas Corona is designed for the Web environment where such properties can vary by several orders of magnitude between objects [10, 19]. Thus, Corona takes object characteristics into account during optimization. Second, the more complex optimization problem renders the Beehive solution technique, based on mathematical derivation, fundamentally unsuitable for the problem tackled by Corona. Hence, Corona employs a more general and sophisticated numerical algorithm to perform its optimizations. Finally, the resource-performance tradeoffs that arise in Corona are fundamentally different from the tradeoffs that Beehive addresses.

3 Corona

Corona (Cornell Online News Aggregator) is a topic-based publish-subscribe system for the Web. It provides asynchronous update notifications to clients while interoperating with the current pull-based architecture of the Web. URLs of Web content serve as topics or *channels* in Corona; users register their interest in some Web content by providing its URL and receive updates asynchronously about changes posted to that URL. Any Web object identifiable by a URL can be monitored with Corona. In the background, Corona checks for updates

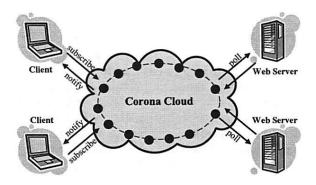


Figure 1: Corona Architecture: Corona is a distributed publish-subscribe system for the Web. It detects Web updates by polling cooperatively and notifies clients through instant messaging.

on registered channels by cooperatively polling the content servers from geographically distributed nodes.

We envision Corona as an infrastructure service offered by a set of widely distributed nodes. These nodes may be all part of the same administrative domain, such as Akamai, or consist of server-class nodes contributed by participating institutions. By participating in Corona, institutions can significantly reduce the network bandwidth consumed by frequent redundant polling for content updates, as well as reduce the peak loads seen at content providers that they themselves may host. Corona nodes self-organize to form a structured overlay system. We use structured overlays to organize the system, as they provide decentralization, good failure resilience, and high scalability [33, 29, 39, 28, 9, 14, 17, 23, 24, 38]. Figure 1 illustrates the overall architecture of Corona.

The key feature that enables Corona to achieve fast update detection is *cooperative polling*. Corona assigns multiple nodes to periodically poll the same channel and shares updates detected by any polling node. In general, n nodes polling with the same polling interval and randomly distributed polling times can detect updates n times faster if they share updates with each other. While it is tempting to take the maximum advantage of cooperative polling by having every Corona node poll for every feed, such a naïve approach is clearly unscalable and would impose substantial network load on both Corona and content servers.

Corona makes informed decisions on distributing polling tasks among nodes. The number of nodes that poll for each channel is determined based on an analysis of the fundamental tradeoff between update performance and network load. Corona poses this tradeoff as an optimization problem and obtains the optimal solution using Honeycomb, a light-weight toolkit for computing optimal performance-overhead tradeoffs in structured distributed systems. This principled approach en-

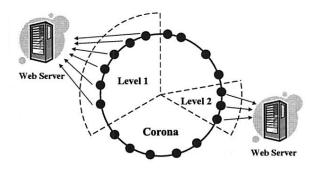


Figure 2: Cooperative Polling in Corona: Each channel is assigned a wedge of nodes to poll the content servers and detect updates. Corona determines the optimal wedge size for each channel through analysis of the global performance-overhead tradeoff.

ables Corona to efficiently resolve the tradeoff between performance and scalability.

In this section, we provide detailed descriptions of the components of Corona's architecture, including the analytical models, optimization framework, update detection and notification mechanisms, and user interface.

3.1 Analytical Modeling

Our analysis-driven approach can be easily applied on any distributed system organized as a structured overlay with uniform node degree. In this paper, we describe Corona using Pastry [29] as the underlying substrate.

Pastry organizes the network into a ring by assigning identifiers from a circular numeric space to each node. The identifiers are treated as a sequence of digits of base b. In addition to neighbors along the ring, each node maintains contact with nodes that have matching prefix digits. These long-distance contacts are represented in a tabular structure called a *routing table*. The entry in the i^{th} row and j^{th} column of a node's routing table points to a node whose identifier shares i prefix digits and whose $(i+1)^{th}$ digit is j. Essentially, the routing table defines a directed acyclic graph (DAG) rooted at each node, enabling a node to reach any other node in $\log_b N$ hops.

Corona assigns nodes in well-defined wedges of the Pastry ring for polling each channel. Each channel is assigned an identifier from the same circular numeric space. A wedge is as set of nodes sharing a common number of prefix digits with a channel's identifier. A channel with *polling level l* is polled by all nodes with at least l matching prefix digits in their identifiers. Thus, polling level 0 indicates that all the nodes in the system poll for the channel. Figure 2 illustrates the concept of polling levels in Corona.

Assigning well-defined portions of the ring to each channel enables Corona to manage polling efficiently with little overhead. The set of nodes polling for a channel can be represented by just a single number, the

polling level, eliminating the expensive O(n) complexity for managing state about cooperating nodes. Moreover, this also facilitates efficient update sharing, as a wedge is a subset of the DAG rooted at each node, and all the nodes in a wedge can be reached quickly using the contacts in the routing table.

The polling level of a channel quantifies its performance-overhead tradeoff. A channel at level l has, on average, $\frac{N}{h!}$ nodes polling it, which can cooperatively detect updates in $\frac{\tau}{2} \frac{b^l}{N}$ time on average, where τ is the polling interval. We estimate the average update detection time at a single node polling periodically at an interval τ to be $\frac{\tau}{2}$. Simultaneously, the collective load placed on the content server of the channel is $\tau \frac{N}{h!}$. Note that we do not include the propagation delay for sharing updates in this analysis because updates can be detected by comparing against any old version of the content. Hence, even if an update detected at a different node in the system is received late, the time to detect the next update at the current node does not change.

An easy way to set polling levels is to independently choose a level for each channel based on these estimates. However, such an approach involves investigating heuristics for determining the appropriate performance requirement for each channel and for dividing the total load between different channels. Moreover, it does not provide fine-grained control over the performance of the system, often causing it to operate far from optimally. The rest of this section describes how the tradeoffs can be posed as mathematical optimization problems to achieve different performance requirements.

Corona-Lite: The first performance goal we explore is minimizing the average update detection time while bounding the total network load placed on content servers. Corona-Lite improves the update performance seen by the clients while ensuring that the content servers handle a light load: no more than they would handle from the clients if the clients fetched the objects directly from the servers.

The optimization problem for Corona-Lite is defined in Table 1. The overall update performance is measured by taking an average of the update detection time of each channel weighted by the number of clients subscribed to the channels. We weigh the average using the number of subscriptions because update performance is an end user experience, and each client counts as a separate unit in the average. The target network load for this optimization is simply the total number of subscriptions in the system.

Corona-Lite clients experience the maximum benefits of cooperation. Clients of popular channels gain greater benefits than clients of less popular channels. Yet, Corona-Lite does not suffer from "diminishing returns," using its surplus polling capacity on less popular channels where the extra bandwidth yields higher marginal benefit. Since improvement in update performance is inversely proportional to the number of polling nodes, a naïve heuristic-based scheme that assigns polling nodes in proportion to number of subscribers would clearly suffer from diminishing returns. Corona, on the other hand, distributes the surplus load to other, less popular channels, improving their update detection times and achieving a better global average.

Corona-Lite:

$$\label{eq:min. state of the min. state of the min. state of the min. } \sum_{1}^{M}q_{i}\frac{b^{l_{i}}}{N} \qquad \qquad \text{s.t. } \sum_{1}^{M}s_{i}\frac{N}{b^{l_{i}}} \quad \leq \quad \sum_{1}^{M}q_{i}$$

Minimize average update detection time, while bounding the load placed on content servers.

Corona-Fast:

min.
$$\sum_{1}^{M} s_i \frac{N}{b^{l_i}}$$
 s.t. $\sum_{1}^{M} q_i \frac{b^{l_i}}{N} \leq T \sum_{1}^{M} q_i$

Achieve a target average update detection time, while minimizing the load placed on content servers.

min.
$$\sum_{1}^{M} q_i \frac{t}{u_i} \frac{b^{l_i}}{N}$$
 s.t. $\sum_{1}^{M} s_i \frac{N}{b^{l_i}} \leq \sum_{1}^{M} q_i$

Minimize average update detection time w.r.t. expected update frequency, bounding load on content servers.

$$\begin{array}{lll} \textbf{Corona-Fair-Sqrt:} \\ \text{min.} \ \, \sum_1^M q_i \frac{\sqrt{\tau}}{\sqrt{u_i}} \frac{b^{l_i}}{N} & \text{s.t.} \ \, \sum_1^M s_i \frac{N}{b^{l_i}} & \leq & \sum_1^M q_i \end{array}$$

Corona-Fair with sqrt weight on the latency ratio to emphasize infrequently changing channels.

Corona-Fair-Log: min.
$$\sum_{1}^{M} q_{i} \frac{\log \tau}{\log u_{i}} \frac{b^{l_{i}}}{N}$$
 s.t. $\sum_{1}^{M} s_{i} \frac{N}{b^{l_{i}}} \leq \sum_{1}^{M} q_{i}$

Corona-Fair with log weight on the latency ratio to emphasize infrequently changing channels.

Notation

polling interval

Mnumber of channels

N number of nodes

b base of structured overlay

Tperformance target

 l_i polling level of channel i

number of clients for channel i q_i

content size for channel i

update interval for channel i

Table 1: Performance-Overhead Tradeoffs: This table summarizes the optimization problems for different performance goals in Corona.

Corona-Fast: While Corona-Lite bounds the network load on the content servers and minimizes update latency, the update performance it provides can vary depending on the current workload. Corona-Fast provides stable update performance, which can be maintained steadily at a desired level through changes in the workload. Corona-Fast solves the converse of the previous optimization problem; that is, it minimizes the total network load on the content servers while meeting a target average update detection time. Corona-Fast enables us to tune the update performance of the system according to application needs. For example, a stock-tracker application may choose a target of 30 seconds to quickly detect changes to stock prices.

Corona-Fast shields legacy Web servers from sudden increases in load. A sharp increase in the number of subscribers for a channel does not trigger a corresponding increase in network load on the Web server since Corona-Fast does not increase polling after diminishing returns sets in. In contrast, in legacy RSS, popularity spikes cause a significant increase in network load on content providers. Moreover, the increased load typically continues unabated in legacy RSS as subscribers forget to unsubscribe, creating "sticky" traffic. Corona-Fast protects content servers from flash-crowds and sticky traffic.

Corona-Fair: Corona-Fast and Corona-Lite do not consider the actual rate of change of content in a channel. While some Web objects are updated every few minutes, others do not change for days at a time [10, 19]. Corona-Fair incorporates the update rate of channels into the performance tradeoff in order to achieve a fairer distribution of update performance between channels. It defines a modified update performance metric as the ratio of the update detection time and the update interval of the channel, which it minimizes to achieve a target load.

While the new metric accounts for the wide difference in update characteristics, it biases the performance unfavorably against channels with large update interval times. A channel that does not change for several days experiences long update detection times, even if there are many subscribers for the channel. We compensate for this bias by exploring other update performance metrics based on square root and logarithmic functions, which grow sublinearly. A sub-linear metric dampens the tendency of the optimization algorithm to punish slow-changing yet popular feeds. Table 1 summarizes the optimization problems for different versions of Corona.

3.2 Decentralized Optimization

Corona determines the optimal polling levels using the Honeycomb optimization toolkit. Honeycomb provides numerical algorithms and decentralized mechanisms for solving optimization problems that can be expressed as follows:

min.
$$\sum_{1}^{M} f_i(l_i)$$
 s.t. $\sum_{1}^{M} g_i(l_i) \leq T$.

Here, $f_i(l)$ and $g_i(l)$ can define the performance or the cost for channel i as a function of the polling level l.

The preceding optimization problem is NP-Hard, as the polling levels only take integral values. Hence, instead of using computationally intensive techniques to find the exact solution, Honeycomb finds an approximate solution quickly in time comparable to a sorting algorithm. Honeycomb's optimization algorithm runs in $O(M\log M\log N)$ time.

The solution provided by Honeycomb is accurate and deviates from the optimal in at most one channel. Honeycomb achieves this accuracy by finding two solutions that optimize the problem with slightly altered constraints: one with a constraint $T_d \leq T$ and another with constraint $T_u \geq T$. The corresponding solutions L_u^* and L_d^* are exactly optimal for the optimization problems with constraints T_u and T_d respectively, and differ in at most one channel. That is, one channel has a different polling level in L_u^* than in L_d^* . Note that the optimal solution L^* for the original problem with constraint T may actually decide to allocate channels differently from L_d^* and L_u^* . Yet, the minimum determined by L^* will be bounded by the minima determined by L_d^* and L_u^* due to monotonicity. Honeycomb then chooses L_d^* as the final solution because it satisfies the constraint T strictly.

Honeycomb computes L_d^* and L_u^* using a Lagrange multiplier to transform the optimization problem as follows:

$$L^* = \arg\min \sum_{1}^{M} f_i(l_i) - \lambda [\sum_{1}^{M} g_i(l_i) - T].$$

Honeycomb iterates over λ and obtains the two solutions L_d^* and L_u^* that bracket the minimum using standard bracketing methods for function optimization in one dimension.

Two observations enable Honeycomb to speed up the optimization algorithm. First, $L^*(\lambda')$ for a single iteration can be computed by finding arg $\min.f_i(l_i)-\lambda'g_i(l_i)$ independently for each channel. This takes $O(M\log N)$ time as the number of levels is bounded by $\lceil \log N \rceil$. Second, for each channel there are only $\log N$ values of λ that change arg $\min.f_i(l_i)-\lambda'g_i(l_i)$. Precomputing these λ values for each object provides a discrete iteration space of $M\log N$ λ values. By keeping a sorted list of the λ values, Honeycomb computes the optimal solution in $O(\log M)$ iterations. Overall, the run-time complexity of the optimization algorithm is $O(M\log M\log N)$ time, including the time spent in pre-computation, sorting, and iterations.

The preceding algorithm requires the tradeoff functions $f_i(l)$ and $g_i(l)$ of all channels in the system in order to compute the global optimum. Solving the optimization problem using limited data available locally can produce highly inaccurate solutions. On the other hand, collecting the tradeoff factors for all the channels

at each node is clearly expensive and impractical. It is possible to gather the tradeoff data at a central node, run the optimization algorithm at a single location, and then distribute the optimal levels to peers from the central location. We avoid using a centralized infrastructure as it introduces a single point of failure in the system and has limited scalability.

Instead, Honeycomb internally aggregates coarse grained information about global tradeoff factors. It combines channels with similar tradeoff factors into a tradeoff cluster. Each cluster summarizes the tradeoff factors for multiple channels and provides coarsegrained tradeoff information. A ratio of performance and cost factors, f_i/g_i , is used as a metric to combine channels. For example, channels with comparable values for $\frac{q_i}{u_i s_i}$ are combined into a cluster in Corona-Fair.

Honeycomb nodes periodically exchange the clusters with contacts in the routing table and aggregate the clusters received from the contacts. Honeycomb keeps the overhead for cluster aggregation low by limiting the number of clusters for each polling level to a constant *Tradeoff_Bins*. Each node receives Tradeoff_Bins clusters for every polling level from each contact in the routing table. Combined, these clusters summarize the tradeoff characteristics of all the channels in the system. The cluster aggregation overhead in terms of memory state as well as network bandwidth is limited by the size of the routing table, and scales with the logarithm of the system size.

3.3 System Management

Corona is a completely decentralized system, in which nodes act independently, share load, and achieve globally optimal performance through mutual cooperation. Corona spreads load uniformly among the nodes through consistent-hashing [18]. Each channel in Corona has a unique identifier and one or more owner nodes managing it. The identifier is a content-hash of the channel's URL, and the primary owner of a channel is the Corona node with the numerically closest identifier to the channel's identifier. Corona adds additional owners for a channel in order to tolerate failures. These owners are the F closest neighbors of the primary owner along the ring. In the event an owner fails, a new neighbor automatically replaces it.

Owners take responsibility for managing subscriptions, polling, and updates for a channel. Owners receive subscriptions through the underlying overlay, which automatically routes all subscription requests of a channel to its owner. The owners keep state about the subscribers of a channel and send notifications to them when fresh updates are detected. In addition, owners also keep track of channel-specific factors that affect the performance tradeoffs, namely the number of subscribers, the size

of the content, and the interval at which servers update channel content. The latter is estimated based on the time between updates detected by Corona.

Corona manages cooperative polling through a periodic protocol consisting of an *optimization phase*, a *maintenance phase*, and an *aggregation phase*. In the optimization phase, Corona nodes apply the optimization algorithm on fine-grained tradeoff data for locally polled channels and coarse-grained tradeoff clusters obtained from overlay contacts. In the maintenance phase, changes to polling levels are communicated to peer nodes in the routing table through *maintenance messages*. Finally, the aggregation phase enables nodes to receive new aggregates of tradeoff factors. In practice, the three phases occur concurrently at a node with aggregation data piggy-backed on maintenance messages.

Corona nodes operate independently and make decisions to increase or decrease polling levels locally. Initially, only the owner nodes at level $K = \lceil \log N \rceil$ poll for the channels. If an owner decides to lower the polling level to K-1 (based on local optimization), it sends a message to the contacts in its routing table at row K-1 in the next *maintenance phase*. As a result, a small wedge of level K-1 nodes start polling for that channel. Subsequently, each of these nodes may independently decide to further lower the polling level of that channel. Similarly, if an owner node decides to raise the level from K-1 to K, it asks its contact in the K-1 wedge to stop polling.

In general, when a level i node lowers the level to i-1 or raises the level from i-1 back to i, it instructs its contact in row i-1 of its routing table to start or stop polling for that channel. This control path closely follows the DAG rooted at the owner node. Nodes at level i (depth K-i) in this DAG decide whether their children at level i-1 should poll a channel and convey these decisions periodically every *maintenance interval*. When a node is instructed to begin polling for a channel, it waits for a random interval of time between 0 and the polling interval before the first poll. This ensures that polls for a channel at different nodes are randomly distributed over time.

Corona nodes gather current estimates of tradeoff factors in the aggregation phase. Owners monitor the number of subscribers and send out fresh estimates along with the maintenance message. Subsequent maintenance messages sent out by descendant nodes in the DAG propagate these estimates to all the nodes in the wedge. The update interval and size of a feed only change during updates and are therefore sent along with updates. Tradeoff clusters are also sent by contacts in the routing table in response to maintenance messages.

Corona inherits robustness and failure-resilience from the underlying structured overlay. If the current contact in the routing table fails, the underlying overlay automatically replaces it with another contact. When new nodes join the system or nodes fail, Corona ensures the transfer of subscription state to new owners. A node that is no longer an owner simply erases its subscription state, and a node that becomes a new owner receives the state from other owners of the channel. Simultaneous failure of more than F adjacent nodes poses a problem for Corona, as well as to many other peer-to-peer systems; we assume that F is chosen to make such an occurrence rare. Note that clients can easily renew subscriptions should a catastrophic failure lose some subscription state.

Overall, Corona manages polling using light-weight mechanisms that impose a small, predictable overhead on the nodes and network. Its algorithms do not rely on expensive constructs such as consensus, leader election, or clock synchronization. Networking activity is limited to contacts in the nodes' routing tables.

3.4 Update Dissemination

Updates are central to the operation of Corona; hence, we ensure that they are detected and disseminated efficiently. Corona uses monotonically increasing numbers to identify versions of content. The version numbers are based on content modification times whenever the content carries such a timestamp. For other channels, the primary owner node assigns version numbers in increasing order based on the updates it receives.

Corona nodes share updates only as *deltas*, the differences between old and new content, rather than the entire content. A measurement study on micronews feeds conducted at Cornell shows that the amount of change in content during an update is typically tiny. The study reports that the average update consists of 17 lines of XML, or 6.8% of the content size [19], which implies that a significant amount of bandwidth can be saved through delta-encoding.

A difference engine enables Corona to identify when a channel carries new information that needs to be disseminated to subscribed clients. The difference engine parses the HTML or XML content to discover the core content in the channel, ignoring frequently changing elements such as timestamps, counters, and advertisements. The difference engine generates a delta if it detects an update after isolating the core content. The data in a delta resembles the typical output of the POSIX 'diff' command: it carries the line numbers where the change occurs, the changed content, an indication of whether it is an addition, omission, or replacement, and a version number of the old content to compare against.

When a delta is generated by a node, it shares the update with all other nodes in the channel's polling wedge. To achieve this, the node simply disseminates the delta along the DAG rooted at it up to a depth equal to the

polling level of the channel. The dissemination along the DAG takes place using contacts in the routing table of the underlying overlay. For channels that cannot obtain a reliable modification timestamp from the server, the node detecting the update sends the delta to the primary owner, which assigns a new version number and initiates the dissemination to other nodes polling that channel. Two different nodes may detect a change "simultaneously" and send deltas to the primary owner. The primary owner always checks the current delta with the latest updated version of the content and ignores redundant deltas.

3.5 User Interface

Corona employs instant messaging (IM) as its user interface. Users add Corona as a "buddy" in their favorite instant messaging system; both subscriptions and update notifications are then communicated as instant messages between the users and Corona. Users send request messages of the form "subscribe url" and "unsubscribe url" to subscribe and unsubscribe for a channel. A subscribe or unsubscribe message delivered by the IM system to Corona is routed to all the owner nodes of the channel, which update their subscription state. When a new update is detected by Corona, the current primary owner sends an instant message with the delta to all the subscribers through the IM system. If a subscriber is offline at the time an update is generated, the IM system buffers the update and delivers it when the subscriber subsequently joins the network.

Delivering updates through instant messaging systems incurs additional latency since messages are sent through a centralized service. However, the additional latency is modest as IM systems are designed to reduce such latencies during two-way communication. Moreover, IM systems that allow peer-to-peer communication between their users, such as Skype, deliver messages in quick time.

Instant messaging enables Corona to be easily accessible to a large user population, as no computer skills other than an ability to "chat" are required, and ubiquitous IM deployment ensures that hosts behind NATs and firewalls are supported. Moreover, instant messages also guarantee the authenticity of the source of update messages to the clients, as instant messaging systems pre-authenticate Corona as the source through password verification.

4 Implementation

We have implemented a prototype of Corona as an application layered on Pastry [29], a prefix-matching structured overlay system. The implementation uses a 160-bit SHA-1 hash function to generate identifiers for both the nodes (based on their IP addresses) and channels (based on their URLs). Both the base of Pastry and the number of tradeoff clusters per polling level are set to 16.

Prefix matching overlays occasionally create *orphans*, that is, channels with no owners having enough matching prefix digits. Orphans are created when the required wedge of the identifier space, corresponding to level $\lceil \log N \rceil - 1$, is empty. Corona cannot assign additional nodes to poll an orphan channel to improve its update performance. Moreover, orphans adversely affect the computation of globally optimal allocation. Corona handles orphan channels by adjusting the tradeoffs appropriately. The tradeoff factors of orphan channels are aggregated into a *slack cluster*, which is used to adjust the performance target prior to optimization.

Corona's reliance on IM as an asynchronous communication mechanism poses some operational challenges. Corona interacts with IM systems using GAIM [12], an open source instant messaging client for Unix-based platforms that supports multiple IM systems including Yahoo Instant Messenger, AOL Instant Messenger, and MSN Messenger. Several IM systems have a limitation that only one instance of a user can be logged on at a time, preventing all Corona nodes from being logged on simultaneously. While we hope that IM systems will support simultaneous logins from automated users such as Corona in the near future, as they have for several chat robots, our implementation uses a centralized server to talk to IM systems as a stop-gap measure. This server acts as an intermediary for all updates sent to clients as well as subscription messages sent by clients. Also, IM systems such as Yahoo rate-limit instant messages sent by unprivileged clients. Corona's implementation limits the rate of updates sent to clients to avoid sending updates in bursts.

Corona trusts the nodes in the system to behave correctly and generate authentic updates. However, it is possible that in a collaborative deployment, where nodes under different administrative domains are part of the Corona network, some nodes may be malicious and generate spurious updates. This problem can be easily solved if content providers are willing to publish digitally signed certificates along with their content. An alternative solution that does not require changes to servers is to use threshold-cryptography to generate a certificate for content [40, 15]. The responsibility for generating partial signatures can be shared among the owners of a node ensuring that rogue nodes below the threshold level cannot corrupt the system. Designing and implementing such a threshold-cryptographic scheme is, however, beyond the scope of this paper.

5 Evaluation

We evaluate the performance of Corona through largescale simulations and wide-area experiments on Planet-Lab [2]. In all our evaluations, we compare the performance of Corona to the performance of legacy RSS, a widely-used micronews syndication system. The simulations and experiments are driven by real-life RSS traces.

We collected characteristics of micronews workloads and content by passively logging user activity and actively polling RSS feeds [19]. User activity recorded between March 22 and May 3 of 2005 at the gateway of the Cornell University Computer Science Department provided a workload of 158 clients making approximately 62,000 requests for 667 different feeds. The channel popularity closely follows a Zipf distribution with exponent 0.5. The survey analyzes the update rate of micronews content by actively polling approximately 100,000 RSS feeds obtained from syndic8.com. We poll these feeds at one hour intervals for 84 hours, and subsequently select a subset of 1000 feeds and poll them at a finer granularity of 10 minutes for five days. Comparing periodic snapshots of the feeds shows that the update interval of micronews content is widely distributed: about 10% of channels changed within an hour, while 50% of channels did not change at all during the five days of polling.

5.1 Simulations

We use tradeoff parameters based on the RSS survey in our simulations. In order to scale the workload to the larger scale of our simulations, we extrapolate the distribution of feed popularity from the workload traces and set the popularity to follow a Zipf distribution with exponent 0.5. We use a distribution for the update rates of channels obtained through active polling, setting the update interval of the channels that do not see any updates to one week.

We perform simulations for a system of 1024 nodes, 100,000 channels, and 5,000,000 subscriptions. We start each simulation with an empty state and issue all subscriptions at once before collecting performance data. We run the simulations for six hours with a polling interval of 30 minutes and maintenance interval of one hour. We study the performance of the three schemes, Corona-Lite, Corona-Fast, and Corona-Fair, and compare the performance with that of legacy RSS clients polling at the same rate of 30 minutes.

Corona-Lite

Figures 3 and 4 show the network load and update performance, respectively, for Corona-Lite, which minimizes average update detection time while bounding the total load on content servers. The figures plot the network load, in terms of the average bandwidth load placed on content servers, and update performance, in terms of the average update detection time. Figure 3 shows that Corona-Lite stabilizes at the load imposed by legacy RSS clients. Starting from a clean slate where only owner

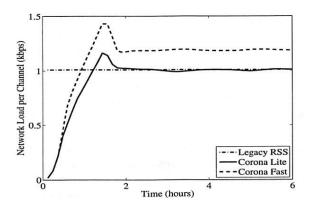


Figure 3: Network Load on Content Servers: Corona-Lite converges quickly to match the network load imposed by legacy RSS clients.

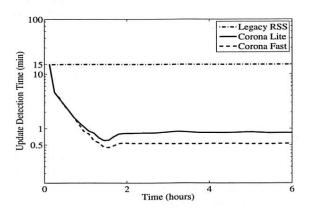


Figure 4: Average Update Detection Time: Corona-Lite provides 15-fold improvement in update detection time compared to legacy RSS clients for the same network load.

nodes poll for each channel, Corona-Lite quickly converges to its target in two maintenance phases. The average load exceeds the target for a brief period before stabilization. This slight delay is due to nodes not having complete information about tradeoff factors of other channels in the system. However, the discrepancy is corrected automatically when aggregated global tradeoff factors are available to each node.

At the same time, Figure 4 shows that Corona-Lite achieves an average update detection time of about one minute. The update performance of Corona-Lite represents an order of magnitude improvement over the average update detection time of 15 minutes provided by legacy RSS clients. This substantial difference in performance is achieved through judicious distribution of polling load between cooperating nodes, while imposing no more load on the servers than the legacy clients.

Figures 5 and 6 show the number of polling nodes assigned by Corona-Lite to different channels and the resulting distribution of update detection times. The x-

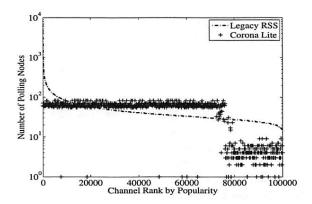


Figure 5: Number of Pollers per Channel: Corona trades off network load from popular channels to decrease update detection time of less popular channels and achieve a lower system-wide average.

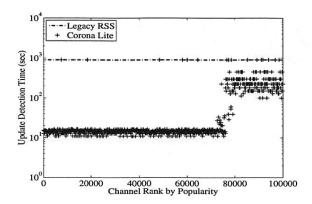


Figure 6: Update Detection Time per Channel: Popular channels gain greater decrease in update detection time than less popular channels.

axis shows channels in reverse order of popularity. We only plot 20,000 channels for clarity. The load imposed by legacy RSS is equal to the number of clients. For Corona-Lite, three levels of polling can be identified in Figure 5: channels clustered around 100 at level 1, channels with fewer than 10 clients at level 2, and orphan channels close to the X-axis with just one owner node polling them. The sharp change in the distribution after 75,000 channels indicates the point where the optimal solution changes polling levels.

Figure 5 shows that Corona-Lite favors popular channels over unpopular ones when assigning polling levels. Yet, it significantly reduces the load on servers of popular content compared to legacy clients, which impose a highly skewed load on content servers and overload servers of popular content. Corona-Lite reduces the load of the over-loaded servers and transfers the extra load to servers of less popular content to improve update performance.

Scheme	Average Update Detection Time (sec)	Average Load (polls per 30 min per channel)		
Legacy-RSS	900	50.00		
Corona-Lite	53	48.97		
Corona-Fair	142	50.14		
Corona-Fair-Sqrt	55	49.46		
Corona-Fair-Log	53	49.43		
Corona-Fast	32	58.75		

Table 2: Performance Summary: This table provides a summary of average update detection time and network load for different versions of Corona. Overall, Corona provides significant improvement in update detection time compared to Legacy RSS, while placing the same load on servers.

The favorable behavior of Corona-Lite is due to diminishing returns caused by the inverse relation between the update detection time and the number of polling nodes. It is more beneficial to distribute the polling across many channels than to devote a large percentage of the bandwidth to polling the most popular channels. Nevertheless, load distribution in Corona-Lite respects the popularity distribution of channels: popular channels are polled by more nodes than less popular channels (see Figure 5). The upshot is that popular channels gain an order of magnitude improvement in update performance over less popular ones (see Figure 6).

Corona-Fast

Unlike Corona-Lite, Corona-Fast minimizes the total load on servers while aiming to achieve a target update detection latency. Figures 3 and 4 show the network load and update performance, respectively, for Corona-Fast. Figure 4 confirms that Corona-Fast closely meets the desired target of 30 seconds. This improvement in update detection time entails an increase in server load over Corona-Lite. Unlike Corona-Lite, whose update performance may vary depending on the workload seen by the system, Corona-Fast provides a stable average update performance. Moreover, it enables us to set the performance depending on the requirements of the application and ensures that the targeted performance is achieved with minimal load on content servers.

Corona-Fair

Finally, we examine the performance of Corona-Fair, which uses the update rates of channels to fine-tune the distribution of load. It takes advantage of the wide distribution of update intervals among channels and aims to poll frequently updated channels at a higher rate than channels with long update intervals. Figure 7 shows the distribution of update detection times achieved by Corona-Lite and Corona-Fair for different channels

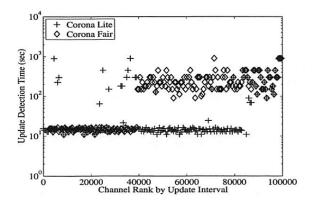


Figure 7: Update Detection Time per Channel: Corona-Fair provides better update detection time for channels that change rapidly than for channels that change rarely.

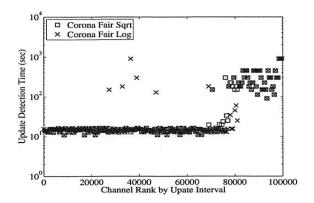


Figure 8: Update Detection Time per Channel: Corona-Fair-Sqrt and Corona-Fair-Log fix the bias against channels that change rarely and provide better update detection time for them than Corona-Fair does.

ranked by their update intervals. Channels with the same update intervals are further ranked by popularity. For clarity of presentation, we plot the distribution for 200 uniformly chosen channels.

Figure 7 shows that Corona-Lite achieves an unfair distribution of update detection times by ignoring update interval information. Many channels with large update intervals have short update detection times (shown in the lower-right of the graph), while some rapidly changing channels have long update detection times (shown in the upper-left of the graph). Corona-Fair fixes this unfair distribution of update detection time by using update intervals of channels to influence the choice of polling levels. Figure 7 shows that Corona-Fair has a fairer distribution of update detection times with update intervals; that is, channels with shorter update intervals have faster update detection times and vice-versa.

Corona-Fair optimizes for update performance measured as the ratio of update detection time and update in-

terval. Thus, channels with long update intervals may also have proportionally long update detection times, leading to long wait times for clients. Section 3.1 proposed to compensate for this bias using two metrics with sub-linear growth based on the square root and logarithm of the update interval. Figure 8 shows that Corona-Fair-Sqrt and Corona-Fair-Log achieve update detection times that are fairer and lower than Corona-Fair. Between the two metrics, Corona-Fair-Sqrt is better than Corona-Fair-Log, which has a few channels with small update intervals but long update detection times.

Overall, the Corona-Fair schemes provide fair distributions of polling between channels with low average update detection times without exceeding bandwidth load on the servers. The average update detection time and load for different Corona-Fair schemes is shown in Table 2. The average update detection time suffers a little in Corona-Fair compared to Corona-Lite, but the modified Corona-Fair schemes provide an average performance close to that of Corona-Lite.

5.2 Deployment

We deployed Corona on a set of 60 PlanetLab nodes and measured its performance. The deployment is based on the Corona-Lite scheme, which minimizes update detection time while bounding network load. For this experiment, we use 7500 real channels providing RSS feeds obtained from www.syndic8.com. We issue 150,000 subscriptions for them based on a Zipf popularity distribution with exponent 0.5. Subscriptions are issued at a uniform rate during the first hour and a half of the experiment. The maintenance interval and the polling interval are both set to 30 minutes. We collected data for a period of six hours.

Figure 9 shows the average update detection time for Corona deployment compared to legacy RSS. Corona decreases the average update time to about 45 seconds compared to 15 minutes for legacy RSS. Figure 10 shows the corresponding polling load imposed by Corona on content servers. Corona gradually increases the number of nodes polling each channel and reaches a load limit of around 4500 polls per minute. Corona's total network load is bounded by the load imposed by legacy RSS, which averages to just above 5000 polls per minute. These graphs highlight that while imposing comparable load as legacy RSS, Corona achieves a substantial improvement in update detection time.

5.3 Summary

The results from simulations and wide-area experiments confirm that Corona achieves a balance between update latency and network load. It dynamically learns the parameters of the system such as number of nodes, number of subscriptions, and tradeoff factors of all channels, and

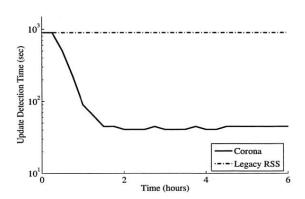


Figure 9: Average Update Detection Time: Corona provides an order of magnitude lower update detection time compared to legacy RSS.

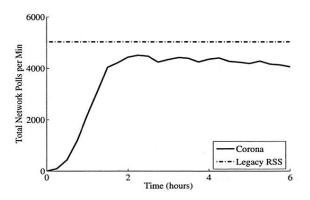


Figure 10: Total Polling Load on Servers: The total load generated by Corona is well below the load generated by clients using legacy RSS

uses the new parameters to periodically adjust the optimal polling levels of channels and meet performance and load targets. Corona offers considerable flexibility in the kind of performance goals it can achieve. In this section, we showed three specific schemes targeting update detection time, network load, and fair distribution of load under different metrics of fairness. Measurements from the deployment showed that achieving globally optimal performance in a distributed wide-area system is practical and efficient. Overall, Corona proves to be a high performance, scalable publish-subscribe system.

6 Conclusions

This paper proposes a novel publish-subscribe architecture that is compatible with the existing pull-based architecture of the Web. Motivated by the growing demand for micronews feeds and the paucity of infrastructure to provide asynchronous notifications, we develop a unique solution that addresses the shortcomings of pull based con-

tent dissemination and delivers a real, deployable, easy-to-use publish-subscribe system.

Many real-world applications require quick and efficient dissemination of information from data sources to clients. It is quite common for legacy data sources, such as Web pages, sensors, stock feeds, event trackers and so forth, to be deployed piecemeal, and thus to force clients to poll them manually and explicitly to receive the latest updates. As the numbers of data sources increase, the task of monitoring so many event sources quickly becomes overwhelming for humans. At sufficiently large scales, the task of allocating bandwidth is difficult even for computers. We can see examples of such applications in large scale sensor networks, in investment management systems that track commodity prices, and in many adaptive distributed systems for detecting events. All of these applications pose a fundamental tension between the polling bandwidth required to achieve fast event detection and the corresponding load imposed by periodic polling.

Our unique contribution is the optimal resolution of performance-overhead tradeoffs in such event detection systems. This paper provides a general approach based on analytical modeling of the cost-performance tradeoff and mathematical optimization that enables applications to make informed, near-optimal decisions on which data sources to monitor, and with what frequency. We develop techniques to solve typical resource allocation problems that arise in distributed systems through decentralized, low-overhead mechanisms.

The techniques at the core of this system are easily applicable to any domain where a set of nodes monitor exogenous events. The Corona approach to monitoring oblivious, pull-based data sources makes it unnecessary to change the data publishing workflow, agree on new dissemination protocols, or deploy new software on data sources. This is particularly relevant when the sources to be monitored are large in number, and deploying new software is logistically difficult. For instance, large scale Web spiders that monitor changes to Websites to incrementally update a Web index could benefit from the principled approach developed here.

Corona applies this general approach to disseminating updates to the Web, where the resource-performance tradeoff is affected by the popularity, size, and update rate of Web content and the network capacities of clients and content servers. Performance measurements based on simulations and real-life deployment show that Corona clients can achieve several orders of magnitude improvement in update latency without an increase in average load. Corona acts as a buffer between clients and servers, shielding servers from the impact of flash-crowds and sticky traffic. Our implementation is currently deployed on PlanetLab and available for pub-

lic use. We hope that a backwards-compatible, highperformance, efficient publish-subscribe system will make it possible for people to easily track frequently changing content on the Web.

Acknowledgements

We would like to thank Rohan Murty for implementing an earlier prototype of the Corona system, Yee Jiun Song for his help with the system, and our shepherd Dave Andersen for his guidance and feedback. This work was supported in part by NSF CAREER grant 0546568, and TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (CCF-0424422) and the following organizations: Cisco, ESCHER, HP, IBM, Intel, Microsoft, ORNL, Qualcomm, Pirelli, Sun, and Symantec.

References

- Atom. Atom Syndication Format. http://www.atomenabled.org/developers/syndication, Oct. 2005.
- [2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of Symposium on Networked Systems Design and Implementation*, Boston, MA, Mar. 2004.
- [3] C. Botev and J. Shanmugasundaram. Context Sensitive Keyword Search and Ranking for XML. In *Proc. of International Workshop on Web and Databases*, Baltimore, MD. June 2005.
- [4] L. F. Cabera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc.* of Workshop on Hot Topics in Operating Systems, Elmau, Germany, May 2001.
- [5] N. Carriero and D. Gelernter. Linda in Context. Communications of the ACM, 32(4):444–458, Apr. 1989.
- [6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3):332–383, Aug. 2001.
- [7] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of International Conference on Data Engineering*, San Jose, CA, Feb. 2002.
- [8] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of Inter*national Conference on Very Large Databases (VLDB), Toronto, Canada, Aug. 2004.
- [9] J. R. Douceur, A. Adya, W. J.Bolosky, and D. Simon. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proc. of International Conference* on *Distributed Computing Systems*, Vienna, Austria, July 2002.
- [10] F. Douglis, A. Feldman, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In Proc. of USENIX Symposium on

- Internet Technologies and Systems, Monterey, CA, Dec. 1997.
- [11] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying (Synopsis). In *Proc. of International Workshop on Web Content Caching and Distribution*, Sophia Antipolis, France, Sept. 2005.
- [12] GAIM. A Multi-Protocol Instant Messaging Client. http://gaim.sourceforge.net, Oct. 2005.
- [13] B. Glade, K. P. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. *Distributed Systems Engineering*, 1(1):29–36, Sept. 1993.
- [14] N. Harvey, M. Jons, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USENIX Symposium* on *Internet Technologies and Systems*, Seattle, WA, Mar. 2003.
- [15] W. K. Josephson, E. G. Sirer, and F. B. Schneider. Peer-to-Peer Authentication With a Distributed Single Sign-On Service. In *Proc. of International Workshop on Peer-to-Peer Systems*, San Diego, CA, Feb. 2004.
- [16] JS-JavaSpaces Service Specification. http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html, 2002.
- [17] F. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In Proc. of International Workshop on Peer-to-Peer Systems, Berkeley, CA, Feb. 2003.
- [18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM Sympo*sium on Theory of Computing, El Paso, TX, Apr. 1997.
- [19] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews. In *Proc. of ACM Internet Measurement Conference*, Berkeley, CA, Oct. 2005.
- [20] L. Liu, C. Pu, and W. Tang. WebCQ: Detecting and Delivering Information Changes on the Web. In Proc. of the International Conference on Information and Knowledge Management, McLean, VA, Nov. 2000.
- [21] L. Liu, C. Pu, W. Tang, and W. Han. CONQUER: A Continual Query System for Update Monitoring in the WWW. International Journal of Computer Systems, Science and Engineering, 14(2):99–112, 1999.
- [22] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of ACM Symposium on Principles of Distributed Comput*ing, Monterey, CA, Aug. 2002.
- [23] P. Maymounkov and D. Mazieres. Kademlia: A Peer-topeer Information System Based on the XOR Metric. In Proc. of International Workshop on Peer-to-Peer Systems, Cambridge, CA, Mar. 2002.
- [24] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured Superpeer: Leveraging Heterogeneity to Provide Constant-time Lookup. In *Proc. of IEEE Workshop on Internet Applications*, San Francisco, CA, Apr. 2003.

- [25] S. Pandey, K. Dhamdere, and C. Olston. WIC: A General-Purpose Algorithm for Monitoring Web Information Sources. In *Proc. of the Conference on Very Large Data Bases (VLDB)*, Toronto, Canada, Aug. 2004.
- [26] S. Pandey, K. Ramamritham, and S. Chakraborti. Monitoring the Dynamic Web to Respond to Continuous Queries. In *Proc. of the International World Wide Web Conference*, Budapest, Hungary, May 2003.
- [27] V. Ramasubramanian and E. G. Sirer. Beehive: Exploiting Power Law Query Distributions for 0(1) Lookup Performance in Peer-to-Peer Overlays. In Proc. of Symposium on Networked Systems Design and Implementation, San Francisco, CA, Mar. 2004.
- [28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.
- [29] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In Proc. of IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany, Nov. 2001.
- [30] RSS 2.0 Specifications. http://blogs.law.harvard.edu/tech/rss, Oct. 2005.
- [31] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. of International Workshop* on *Peer-to-Peer Systems*, Ithaca, NY, Feb. 2005.
- [32] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin. In *Proc.* of AUUG2K, Canberra, Australia, June 2000.
- [33] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Bal-akrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIG-COMM*, San Diego, CA, Aug. 2001.
- [34] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. of International Symposium on Software* Reliability Engineering, Paderborn, Germany, Nov. 1998.
- [35] TIBCO Publish-Subscribe. http://www.tibco.com, Oct. 2005.
- [36] TSpaces. http://www.almaden.ibm.com/cs/TSpaces/, Oct. 2005.
- [37] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. ACM Transactions on Computer Systems, 21(3), May 2003.
- [38] U. Wieder and M. Naor. A Simple Fault Tolerant Distributed Hash Table. In *Proc. of International Workshop* on *Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.
- [39] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.
- [40] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. ACM Transactions on Computer Systems, 20(4):329–368, Nov. 2002.

Scale and Performance in the CoBlitz Large-File Distribution Service

KyoungSoo Park and Vivek S. Pai Department of Computer Science Princeton University

Abstract

Scalable distribution of large files has been the area of much research and commercial interest in the past few years. In this paper, we describe the CoBlitz system, which efficiently distributes large files using a content distribution network (CDN) designed for HTTP. As a result, CoBlitz is able to serve large files without requiring any modifications to standard Web servers and clients, making it an interesting option both for end users as well as infrastructure services. Over the 18 months that CoBlitz and its partner service, CoDeploy, have been running on PlanetLab, we have had the opportunity to observe its algorithms in practice, and to evolve its design. These changes stem not only from observations on its use, but also from a better understanding of their behavior in real-world conditions. This utilitarian approach has led us to better understand the effects of scale, peering policies, replication behavior, and congestion, giving us new insights into how to better improve their performance. With these changes, CoBlitz is able to deliver in excess of 1 Gbps on PlanetLab, and to outperform a range of systems, including research systems as well as the widely-used BitTorrent.

1 Introduction

Many new content distribution networks (CDNs) have recently been developed to focus on areas not generally associated with "traditional" Web (HTTP) CDNs. These systems often focus on distributing large files, especially in flash crowd situations where a news story or software release causes a spike in demand. These new approaches break away from the "whole-file" data transfer model, the common access pattern for Web content. Instead, clients download pieces of the file (called chunks, blocks, or objects) and exchange these chunks with each other to form the complete file. The most widely used system of this type is BitTorrent [12], while related research systems include Bullet [20], Shark [2], and FastReplica [9].

Using peer-to-peer systems makes sense when the window of interest in the content is short, or when the content provider cannot afford enough bandwidth or CDN hosting costs. However, in other scenarios, a managed CDN service may be an attractive option, espe-

cially for businesses that want to offload their bandwidth but want more predictable performance. The problem arises from the fact that HTTP CDNs have not traditionally handled this kind of traffic, and are not optimized for this workload. In an environment where objects average 10KB, and where whole-file access is dominant, suddenly introducing objects in the range of hundreds of megabytes may have undesirable consequences. For example, CDN nodes commonly cache popular objects in main memory to reduce disk access, so serving several large files at once could evict thousands of small objects, increasing their latency as they are reloaded from disk.

To address this problem, we have developed the CoBlitz large file transfer service, which runs on top of the CoDeeN content distribution network, an HTTP-based CDN. This combination provides several benefits: (a) using CoBlitz to serve large files is as simple as changing its URL – no rehosting, extra copies, or additional protocol support is required; (b) CoBlitz can operate with unmodified clients, servers, and tools like curl or wget, providing greater ease-of-use for users and for developers of other services; (c) obtaining maximum perclient performance does not require multiple clients to be downloading simultaneously; and (d) even after an initial burst of activity, the file stays cached in the CDN, providing latecomers with the cached copy.

From an operational standpoint, this approach of running a large-file transfer service on top of an HTTP content distribution network also has several benefits: (a) given an existing CDN, the changes to support scalable large-file transfer are small; (b) no dedicated resources need to be devoted for the large-file service, allowing it to be practical even if utilization is low or bursty; (c) the algorithmic changes to efficiently support large files also benefit smaller objects.

Over the 18 months that CoBlitz and its partner service, CoDeploy, have been running on PlanetLab, we have had the opportunity to observe its algorithms in practice, and to evolve its design, both to reflect its actual use, and to better handle real-world conditions. This utilitarian approach has given us a better understanding of the effects of scale, peering policies, replication behavior, and congestion, giving us new insights into how to improve performance and reliability. With these

changes, CoBlitz is able to deliver in excess of 1 Gbps on PlanetLab, and to outperform a range of systems, including research systems as well as BitTorrent.

In this paper, we discuss what we have learned in the process, and how the observations and feedback from long-term deployment have shaped our system. We discuss how our algorithms have evolved, both to improve performance and to cope with the scalability aspects of our system. Some of these changes stem from observing the real behavior of the system versus the abstract underpinnings of our original algorithms, and others from observing how our system operates when pushed to its limits. We believe that our observations will be useful for three classes of researchers: (a) those who are considering deploying scalable large-file transfer services; (b) those trying to understand how to evaluate the performance of such systems, and; (c) those who are trying to capture salient features of real-world behavior in order to improve the fidelity of simulators and emulators.

2 Background

In this section, we provide general information about HTTP CDNs, the problems caused by large files, and the history of CoBlitz and CoDeploy.

2.1 HTTP Content Distribution Networks

Content distribution networks relieve Web congestion by replicating content on geographically-distributed servers. To provide load balancing and to reduce the number of objects served by each node, they use partitioning schemes, such as consistent hashing [17], to assign objects to nodes. CDN nodes tend to be modified proxy servers that fetch files on demand and cache them as needed. Partitioning reduces the number of nodes that need to fetch each object from the origin servers (or other CDN nodes), allowing the nodes to cache more objects in main memory, eliminating disk access latency and improving throughput.

In this environment, serving large files can cause several problems. Loading a large file from disk can temporarily evict several thousand small files from the inmemory cache, reducing the proxy's effectiveness. Popular large files can stay in the main memory for a longer period, making the effects more pronounced. To get a sense of the performance loss that can occur, one can examine results from the Proxy Cacheoffs [25], which show that the same proxies, when operating as "Web accelerators," can handle 3-6 times the request rate than operating in "forward mode," with much larger working sets. So, if a CDN node suddenly starts serving a data set that exceeds its physical memory, its performance will drop dramatically, and latency rises sharply. Bruce Maggs, Akamai's VP of Research, states:

"Memory pressure is a concern for CDN developers, because for optimal latency, we want to ensure that the tens of thousands of popular objects served by each node stay in the main memory. Especially in environments where caches are deployed inside the ISP, any increase in latency caused by objects being fetched from disk would be a noticeable degradation. In these environments, whole-file caching of large files would be a concern [21]."

Akamai has a service called EdgeSuite Net Storage, where large files reside in specialized replicated storage, and are served to clients via overlay routing [1]. We believe that this service demonstrates that large files are a qualitatively different problem for CDNs.

2.2 Large-file Systems

As a result of these problems and other concerns, most systems to scalably serve large files departed from the use of HTTP-based CDNs. Two common design principles are evident in these systems: treat large files as a series of smaller chunks, and exchange chunks between clients, instead of always using the origin server. Operating on chunks allows finer-grained load balancing, and avoids the trade-offs associated with large-file handling in traditional CDNs. Fetching chunks from other peers not only reduces load on the origin, but also increases aggregate capacity as the number of clients increases.

We subdivide these systems based on their inter-client communication topology. We term those that rely on greedy selection or all-to-all communication as examples of the *swarm* approach, while those that use tree-like topologies are termed *stream* systems.

Swarm systems, such as BitTorrent [12] and Fast-Replica [9], preceded stream systems, and scaled despite relatively simple topologies. BitTorrent originally used a per-file centralized directory, called a tracker, that lists clients that are downloading or have recently downloaded the file. Clients use this directory to greedily find peers that can provide them with chunks. The newest BitTorrent can operate with tracker information shared by clients. In FastReplica, all clients are known at the start, and each client downloads a unique chunk from the origin. The clients then communicate in an all-to-all fashion to exchange chunks. These systems reduce link stress compared to direct downloads from the origin, but some chunks may traverse shared links repeatedly if multiple clients download them.

The stream systems, such as ESM [10], Split-Stream [8], Bullet [20], Astrolabe [30], and FatNemo [4] address the issues of load balancing and link stress by optimizing the peer-selection process. The result generates a tree-like topology (or a mesh or gossip-based network inside the tree), which tends to stay relatively stable during the download process. The effort in tree-building can

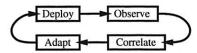


Figure 1: Operational model for CoBlitz improvement

produce higher aggregate bandwidths, suitable for transmitting the content simultaneously to a large number of receivers. The trade-off, however, is that the higher link utilization is possible only with greater synchrony. If receivers are only loosely synchronized and chunks are transmitted repeatedly on some links, the transmission rate of any subtrees using those nodes also decreases. As a result, these systems are best suited for synchronous activity of a specified duration.

2.3 CoBlitz, CoDeploy, and CoDeeN

This paper discusses our experience running two largefile distribution systems, CoBlitz and CoDeploy, which operate on top of the CoDeeN content distribution network. CoDeeN is a HTTP CDN that runs on every available PlanetLab node, with access restrictions in place to prevent abuse and to comply with hosting site policies. It has been in operation for nearly three years, and currently handles over 25 million requests per day. To use CoDeeN, clients configure their browsers to use a CoDeeN node as a proxy, and all of their Web traffic is then handled by CoDeeN. Note that this behavior is only part of CoDeeN as a policy decision – CoBlitz does not require changing any browser setting.

Both CoBlitz and CoDeploy use the same infrastructure, which we call CoBlitz in the rest of this paper for simplicity. The main difference between the two is the access mechanism – CoDeploy requires the client to be a PlanetLab machine, while CoBlitz is publicly accessible. CoDeploy was launched first, and allows PlanetLab researchers to use a local instance of CoDeeN to fetch experiment files. CoBlitz allows the public to access CoDeploy by providing a simpler URL-based interface. To use CoBlitz, clients prepend the original URL with http://coblitz.codeen.org:3125/ and fetch it like any other URL. A customized DNS server maps the name coblitz.codeen.org to a nearby PlanetLab.

In 18 months of operation, the system has undergone three sets of changes: scaling from just North American PlanetLab nodes to all of PlanetLab, changing the algorithms to reduce load at the origin server, and changing the algorithms to reduce overall congestion and increase performance. Our general mode of operation is shown in Figure 1, and consists of four steps: (1) deploy the system, (2) observe its behavior in actual operation, (3) determine how the underlying algorithms, when exposed to the real environment, cause the behaviors, and

(4) adapt the algorithms to make better decisions using the real-world data. We believe this approach has been absolutely critical to our success in improving CoBlitz, as we describe later in this paper.

3 Design of Large File Splitting

Before discussing CoBlitz's optimizations, we first describe how we have made HTTP CDNs amenable to handling large files. Our approach has two components: modifying large file handling to efficiently support them on HTTP CDNs, and modifying the request routing for these CDNs to enable more swarm-like behavior under heavy load. Though we build on the CoDeeN CDN, we do not believe any of these changes are CoDeeN-specific – they could equally be applied to other CDNs. Starting from an HTTP CDN maintains compatibility with standard Web clients and servers, whereas starting with a stream-oriented CDN might require more effort to efficiently support standard Web traffic.

3.1 Requirements

We treat large files as a set of small files that can be spread across the CDN. To make this approach as transparent as possible to clients and servers, the dynamic fragmentation and reassembly of these small files is performed inside the CDN, on demand. Each CDN node has an agent that accepts clients' requests for large files and converts them into a series of requests for pieces of the file. Pieces are specified using HTTP/1.1 byte ranges [14], which the Apache Web server has supported since August 1996 (version 1.2), and which appeared in other servers in the same timeframe. After these requests are injected into the CDN, the results are reassembled by the agent and passed to the client. For simplicity, this agent occupies a different port number than regular CoDeeN requests. The process has some complications, mostly related to the design of traditional CDNs, limitations of HTTP, and the limitations of standard HTTP proxies (which are used as the CDN nodes). Some of these problems include:

Chunk naming – If chunks are named using the original URL, all of a file's chunks will share the same name, and will be routed similarly since CDNs hash URLs for routing [16, 31]. Since we want to spread chunks across the CDN, we must use a different chunk naming scheme.

Range caching – We know of no HTTP proxies that cache arbitrary ranges of Web objects, though some can serve ranges from cached objects, and even recreate a full object from all of its chunks. Since browsers are not likely to ask for arbitrary and disjoint pieces of an object, no proxies have developed the necessary support. Since we want to cache at the chunk level instead of the file level, we must address this limitation.

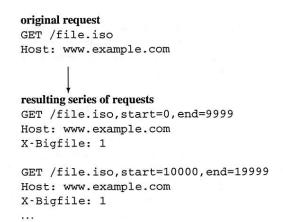


Figure 2: The client-facing agent converts a single request for a large file into a series of requests for smaller files. The new URLS are only a CDN-internal representation – neither the client nor the origin server see them.

Congestion – During periods of bursty demand and heavy synchrony, consistent hashing may produce roving instantaneous congestion. If many clients at different locations suddenly ask for the same file, a lightly-loaded CDN node may see a burst of request. If the clients all ask for another file as soon as the first download completes, another CDN node may become instantly congested. This bursty congestion prevents using the aggregate CDN bandwidth effectively over short time scales.

We address these problems as a whole, to avoid new problems from piecemeal fixes. For example, adding range caching to the Squid proxy has been discussed since 1998 [24], but would expand the in-memory metadata structures, increasing memory pressure, and would require changing the internet cache protocol (ICP) used by caches to query each other. Even if we added this support to CoDeeN's proxies, it would still require extra support in the CDN, since the range information would have to be hashed along with the URL.

3.2 Chunk Handling Mechanics

We modify intra-CDN chunk handling and request redirection by treating each chunk as a real file with its own name, so the bulk of the CDN does not need to be modified. This name contains the start and end ranges of the file, so different chunks will have different hash values. Only the CDN ingress/egress points are affected, at the boundaries with the client and the origin server.

The agent takes the client's request, converts it into a series of requests for chunks, reassembles the responses, and sends it to the client. The client is not aware that the request is handled in pieces, and no browser modifications are needed. This process is implemented in a small program on each CDN node, so communication

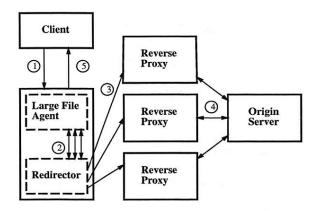


Figure 3: Large-file processing – 1. the client sends the agent a request, 2. the agent generates a series of URL-mangled chunk requests, 3. those requests are spread across the CDN, 4. assuming cache misses, the URLs are de-mangled on egress, and the responses are modified, 5. the agent collects the responses, reassembles if needed, and streams it to the client

between it and the CDN infrastructure is cheap. The requests sent into the CDN, shown in Figure 2, contain extended filenames that specify the actual file and the desired byte range, as well as a special header so that the CDN modifies these requests on egress. Otherwise, these requests look like ordinary requests with slightly longer filenames. The full set of steps are shown in Figure 3, where each solid rectangle is a separate machine connected via the Internet.

All byte-range interactions take place between the proxy and the origin server – on egress, the request's name is reverted, and range headers are added. The server's response is changed from a HTTP 206 code (partial content received) to 200 (full file received). The underlying proxy never sees the byte-range transformations, so no range-caching support is required. Figure 4 shows this process with additional temporary headers. These headers contain the file length, allowing the agent to provide the content length for the complete download.

Having the agent use the local proxy avoids having to reimplement CDN code (such as node liveness, or connection management) in the agent, but can cause cache pollution if the proxy caches all of the agent's requests. The ingress add a cache-control header that disallows local caching, which is removed on egress when the proxy routes the request to the next CDN node. As a result, chunks are cached at the next-hop CDN nodes instead of the local node.

Since the CDN sees a large number of small file requests, it can use its normal routing, replication, and caching policies. These cached pieces can then be used to serve future requests. If a node experiences cache

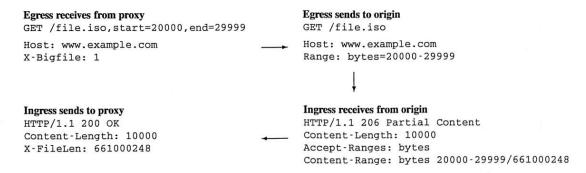


Figure 4: Egress and ingress transformations when the CDN communicates with the origin server. The CDN internally believes it is requesting a small file, and the egress transformation requests a byte-range of a large file. The ingress converts the server's response to a response for a complete small file, rather than a piece of a large file.

pressure, it can evict as many pieces as needed, instead of evicting one large file. Similarly, the addition/departure of nodes will only cause missing pieces to be re-fetched, instead of the whole file. The only external difference is that the server sees byte-range requests from many proxies instead of one large file request from one proxy.

3.3 Agent Design

The agent is the most complicated part of CoBlitz, since it must operate smoothly, even in the face of unpredictable CDN nodes and origin servers outside our control. The agent monitors the chunk downloads for correctness checking and for performance. The correctness checking consists of issues such as ensuring that the server is capable of serving HTTP byte-range requests, verifying that the response is cacheable, and comparing modification headers (file length, last-modified time, etc) to detect if a file has changed at the origin during its download. In the event of problems, the agent can abort the download and return an error message to the client. The agent is the largest part of CoBlitz – it consists of 770 semicolon-lines of code (1975 lines total), versus 60-70 lines of changes for ingress/egress modifications.

To determine when to re-issue chunk fetches, the agent maintains overall and per-chunk statistics during the download. Several factors may slow chunk fetching, including congestion between the proxy and its peers, operational problems at the peers, and congestion between the peers and the origin. After downloading the first chunk, the agent has the header containing the overall file size, and knows the total number of chunks to download. It issues parallel requests up to its limit, and uses non-blocking operations to read data from the sockets as it becomes available.

Using an approach inspired by LoCI [3], slow transfers are addressed by issuing multiple requests – whenever a chunk exceeds its download deadline, the agent opens a new connection and re-issues the chunk request.

The most recent request for the same chunk is allowed to continue downloading, and any earlier requests for the chunk are terminated. In this way, each chunk can have at most two requests for it in flight from the agent, a departure from LoCI where even more connections are made as the deadline approaches. The agent modifies a noncritical field of the URL in retry requests beyond the first retried request for each chunk. This field is stripped from the URL on egress, and exists solely to allow the agent to randomize the peer serving the chunk. In this way, the agent can exert some control over which peer serves the request, to reduce the chance of multiple failures within the CDN. Keeping the same URL on the first retry attempts to reduce cache pollution - in a load-balanced, replicated CDN, the retry is unlikely to be assigned to the same peer that is handling the original request.

The first retry timeout for each chunk is set using a combination of the standard deviation and exponentially-weighted moving average for recent chunks. Subsequent retries use exponential backoff to adjust the deadline, up to a limit of 10 backoffs per chunk. To bound the backoff time, we also have a hard limit of 10 seconds for the chunk timeout. The initial timeout is set to 3 seconds for the first chunk – while most nodes finish faster, using a generous starting point avoids overloading slow origin servers. In practice, 10-20% of chunks are retried, but the original fetch usually completes before the retry. We could reduce retry aggressiveness, but this approach is unlikely to cause much extra traffic to the origin since the first retry uses a different replica with the same URL.

By default, the agent sends completed chunks to the client as soon as they finish downloading, as long as all preceding chunks have also been sent. If the chunk at the head of the line has not completed downloading, no new data is sent to the client until the chunk completes. By using enough parallel chunk fetches, delays in downloading chunks can generally be overlapped with others in the pipeline. If clients that can use chunked transfer encod-

ing provide a header in the request indicating they are capable of handling chunks in any order, the agent sends chunks as they complete, with no head-of-line blocking. Chunk position information is returned in a trailer following each chunk, which the client software can use to assemble the file in the correct order.

The choice of chunk size is a trade-off between efficiency and latency – small chunks will result in faster chunk downloads, so slower clients will have less impact. However, the small chunks require more processing at all stages – the agent, the CDN infrastructure, and possibly the origin server. Larger chunks, while more efficient, can also cause more delay if head-of-line blocking arises. After some testing, we chose a chunk size of 60KB, which is large enough to be efficient, but small enough to be manageable. In particular, this chunk size can easily fit into Linux's default outbound kernel socket buffers, allowing the entire chunk to be written to the socket with a single system call that returns without blocking.

3.4 Design Benefits

We believe that this design has several important features that not only make it practical for deployment now, but will continue to make it useful in the future:

No client synchronization – Since chunks are cached in the CDN when first downloaded, no client synchronization is needed to reduce origin traffic. If clients are highly synchronized, agents can use the same chunk to serve many client requests, reducing the number of intra-CDN transfers, but synchronization is not required for efficient operation.

Trading bandwidth for disk seeks – Fetching most chunks from other CDN nodes trades bandwidth for disk seeks. Given the rate of improvement of each, this trade-off will hold for the foreseeable future. Bandwidth is continually dropping in price, and disk seek times are not scaling. If this bandwidth cost is an issue, it can be billed just as regular bandwidth is billed.

Increasing chunk utility – Having all nodes store chunks makes them available to a larger population than storing the entire file at a small number of nodes. Many more nodes can now serve large files, so the total capacity is the sum of the bandwidths they have to serve clients, and the aggregate intra-CDN capacity is available to exchange chunks.

Using cheaper bandwidth – When CDN nodes communicate with each other, this bandwidth consumption is either within a LAN cluster hosting the CDN nodes, or toward the network core, away from the clients that sit at the edge of the network. Core bandwidth has been improving in

price/performance more rapidly than edge bandwidth, and LAN bandwidth is virtually free, so this consumption is in the more desirable direction.

Scaling with CDN size – As CDN size increases, and aggregate physical memory increases, chunks can be replicated more widely. The net result is that desired chunks are more likely to be in nearby nodes, so link stress drops as the CDN grows.

Tunable memory consumption – Varying the number of parallel chunks downloads that are used for each client controls the memory consumption of this approach. Slower clients can be allocated fewer parallel chunks, and the aggregate number of chunks can be reduced if a node is experiencing heavy load.

In-order or out-of-order delivery – For regular browsers or other standard client software, chunks are delivered in order so that the download appears exactly like a non-CoBlitz download from the origin, and performance-hungry clients can use software that supports chunked encoding.

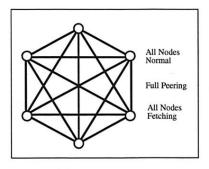
4 Coping With Scale

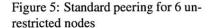
One first challenge for CoBlitz was handling scale – at the time of CoBlitz's original deployment, CoDeeN was running on all 100 academic PlanetLab node in North America. The first major scale issue was roughly quadrupling the node count, to include every PlanetLab node. In the process, we adopted three design decisions that have served us well: (a) make peering a unilateral, asynchronous decision, (b) use minimum application-level ping times when determining suitable peers, and (c) apply hysteresis to the peer set. These are described in the remainder of this section.

4.1 Unilateral, Asynchronous Peering

In CoDeeN, we have intentionally avoided any synchronized communication for group maintenance, which results in avoiding any quorum protocols, 2-phase behavior, or any group membership protocols. The motivations behind this decision were simplicity and robustness – by making every decision unilaterally and independently at each node, we avoid any situation where forward progress fails because some handshaking protocol fails. As a result, CoDeeN has been operational even in some very extreme circumstances, such as in February 2005, when a kernel bug caused the sudden, near-simultaneous failures of nodes, with more than half of all PlanetLab nodes freezing.

One side-effect of asynchronous communication is that all peering is unilateral – nodes independently pick





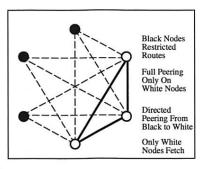


Figure 6: Peering with semiroutable Internet2

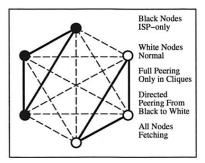


Figure 7: Peering with policy-restricted nodes

their peers, using periodic heartbeats and acknowledgments to judge peer health. Pairwise heartbeats are simple, robust, and particularly useful for testing reachability. More sophisticated techniques, such as aggregating node health information using trees, can reduce the number of heartbeats, but can lead to *worse* information, since the tree may miss or use different links than those used for pairwise communication.

Unilateral and unidirectional peering improves CoBlitz's scalability, since it allows nodes with heterogeneous connectivity or policy issues to participate to the extent possible. These scenarios are shown in Figures 5, 6, and 7. For example, research networks like Internet2 or CANARIE (the Canadian high-speed network) do not peer with the commercial Internet, but are reachable from a number of research sites including universities and corporate labs. These nodes advertise that they do not want any nodes (including each other) using them as peers, since they cannot fetch content from the commercial Internet. These sites can unidirectionally peer with any CoDeeN nodes they can reach - regular CoDeeN nodes do not reciprocate, since the restricted nodes cannot fetch arbitrary Web Also, in certain PlanetLab locations, both corporate and regional, political/policy considerations make the transit of arbitrary content an unwise idea, but the area may have a sizable number of nodes. These nodes advertise that only other nodes from the same organization can use them as peers. These nodes will peer both with each other and with unrestricted nodes, giving them more peers for CoBlitz transfers than they would have available otherwise. Policy restrictions are not PlanetLab-specific - ISPs host commercial CDN nodes in their network with the restriction that the CDN nodes only serve their own customers.

4.2 Peer Set Selection

With the worldwide deployment of CoDeeN, some step had to be taken to restrict the set of CoDeeN nodes that each node would use as peers. Each CoDeeN node sends one heartbeat per second to another node, so at 600+ PlanetLab nodes (of which 400+ are alive at any time), a full sweep would take 10 minutes. Using our earlier measurements that node liveness is relatively stable at short timescales [32], we limit the peer set to 60 nodes, which means that using an additional once-per-second ping, the peers can be swept once per minute.

To get some indication of application health, CoDeeN uses application-level pings, rather than network pings, to determine round trip times (RTTs). Originally, CoDeeN kept the average of the four most recent RTT values, and selected the 60 closest peers within a 100ms RTT cutoff. The 100ms cutoff was to reduce noticeable lag in interactive settings, such as Web browsing. In parts of the world where nodes could not find 20 peers within 100ms, this cutoff is raised to 200ms and the 20 best peers are selected.

This approach exhibited two problems – a high rate of change in the peer sets, and low overlap among peer sets for nearby peers. The high change rate potentially impacts chunk caching in CoBlitz - if the peer that previously fetched a chunk is no longer in the peer set, the new peer that replaces it may not yet have fetched the chunk. To address this issue, hysteresis was added to the peer set selection process. Any peer not on the set could only replace a peer on the set if it was closer in two-thirds of the last 32 heartbeats. Even under the worst-case conditions, using the two-thirds threshold would keep a peer on the set for 10 minutes at a time. While hysteresis reduced peer set churn, it also reinforced the low overlap between neighboring peer sets. Further investigation indicated that CoDeeN's application-level heartbeats had more than an order of magnitude variance than network pings. This variance led to instability in the average RTT calculations, so once nodes were added to the peer set, they rarely got displaced.

Switching from an *average* application-level RTT to the *minimum* observed RTT (an approach also used in other systems [6, 13, 22]) and increasing the number of samples yielded significant improvement, with

∀ nodes, hash(i) = hashcalc(URL, node name(i))
hash = sort(hash)
hash = truncate(hash, NumCandidates)
∀ nodes, index(i) = node index number of hash(i)
minval = min(load(index(i)))
hash = select hash where load(index(i)) == minval
return index(random() modulo size(hash))

Figure 8: Replicated Highest Random Weight with Load Balancing, as used in CoDeeN

application-level RTTs correlating well with ping time on all functioning nodes. Misbehaving nodes still showed large application-level minimum RTTs, despite having low ping times. The overlap of peer lists for nodes at the same site increased from roughly half to almost 90%. At the same time, we discovered that many intra-PlanetLab paths had very low latency, and restricting the peer size to 60 was needlessly constrained. We increased this limit to 120 nodes, and issued 2 heartbeats per second. Of the nodes regularly running CoDeeN, two-thirds tend to now have 100+ peers. More details of the redesign process and its corresponding performance improvement can be found in our previous study [5].

4.3 Scaling Larger

It is interesting to consider whether this approach could scale to a much larger system, such as a commercial CDN like Akamai. By the numbers, Akamai is about 40 times as large as our deployment, at 15,000 servers across 1,100 networks. However, part of what makes scaling to this size simpler is deploying clusters at each network point-of-presence (POP), which number only 2,500. Further, their servers have the ability to issue reverse ARPs and assume the IP addresses of failing nodes in the cluster, something not permitted on PlanetLab. With this ability, the algorithms need only scale to the number of POPs, since the health of a POP can be used instead of querying the status of each server. Finally, by imposing geographic hierarchy and ISP-level restrictions, the problem size is further reduced. With these assumptions, we believe that we can scale to larger sizes without significant problems.

5 Reducing Load & Congestion

Reducing origin server load and reducing CDN-wide congestion are related, so we present them together in this section. Origin load is an important metric for CoBlitz, because it determines CoBlitz's caching benefit and impacts the system's overall performance. From a content provider's standpoint, CoBlitz would fetch only a single copy of the content, no matter what the demand. However, for reasons described below, this goal may not be practical.

5.1 The HRW Algorithm

CoDeeN uses the Highest Random Weight (HRW) [29] algorithm to route requests from clients. This algorithm is functionally similar to Consistent Hashing [17], but has some properties that make it attractive when object replication is desired [31]. The algorithm used in CoDeeN, Replicated HRW with Load Balancing, is shown in Figure 8.

For each URL, CoDeeN generates an array of values by hashing the URL with the name of each node in the peer set. It then prunes this list based on the replication factor specified, and then prunes it again so only the nodes with the lowest load values remain. The final candidate is chosen randomly from this set. Using replication and load balancing reduces hot spots in the CDN – raising the replication factor reduces the chance any node gets a large number of requests, but also increases the node's working set, possibly degrading performance.

5.2 Increasing Peer Set Size

Increasing the peer set size, as described in Section 4.2 has two effects – each node appears as a peer of many more nodes than before, and the number of nodes chosen to serve a particular URL is reduced. In the extreme, if all CDN nodes were in each others' peer sets, then the total number of nodes handling any URL would equal to *NumCandidates*. In practice, the peer sets give rise to overlapping regions, so the number of nodes serving a particular URL is tied to the product of the number of regions and *NumCandidates*.

When examining origin server load in CoBlitz, we found that nodes with fewer than five peers generate almost one-third of the traffic. Some poorly-connected sites have such high latency that even with an expanded RTT criterion, they find few peers. At the same time, few sites use them as peers, leading to them being an isolated cluster. For regular Web CDN traffic, these small clusters are not much of an issue, but for large-file traffic, the extra load these clusters cause on the origin server slows the rest of the CDN significantly. Increasing the minimum number of peers per node to 60 reduces traffic to the origin. Because of unilateral peering, this change does not harm nearby nodes – other nodes still avoid these poorly-connected nodes.

Reducing the number of replicas per URL reduces origin server load, since fewer nodes fetch copies from the origin, but it also causes more bursty traffic at those replicas if downloading is synchronized. For CoBlitz, synchronized downloads occur when developers push software updates to all nodes, or when cron-initiated tasks simultaneously fetch the same file. In these cases, demand at any node experiences high burstiness over short time scales, which leads to congestion in the CDN.

5.3 Fixing Peer Set Differences

Once other problems are addressed, differences in peer sets can also cause a substantial load on the origin server. To understand how this arises, imagine a CDN of 60 nodes, where each node does not see one peer at random. If we ask all nodes for the top candidate in the HRW list for a given URL, at least one node is likely to return the candidate that would have been the second-best choice elsewhere. If we ask for the top k candidates, the set will exceed k candidates with very high probability. If each node is missing two peers at random, the union of the sets is likely to be at least k+2. Making the matter worse is that these "extra" nodes fetching from the origin also provide very low utility to the rest of the nodes – since few nodes are using them to fetch the chunk, they do not reduce the traffic at the other replicas.

To fix this problem, we observe that when a node receives a forwarded request, it can independently check to see whether it should be the node responsible for serving that request. On every forwarded request that is not satisfied from the cache, the receiving node performs its own HRW calculation. If it finds itself as one of the top candidates, it considers the forwarded request reasonable and fetches it from the origin server. If the receiver finds that it is not one of the top candidates, it forwards the request again. We find that 3-7% of chunks get re-forwarded this way in CoBlitz, but it can get as high as 10-15% in some cases. When all PlanetLab nodes act as clients, this technique cuts origin server load almost in half.

Due to the deterministic order of HRW, this approach is guaranteed to make forward progress and be loop-free. While the worst case is a number of hops linear in the number of peer groups, this case is also exponentially unlikely. Even so, we limit this approach to only one additional hop in the redirection, to avoid forwarding requests across the world and to limit any damage caused by bugs in the forwarding logic. Given the relatively low rate of chunks forwarded in this manner, restricting it to only one additional hop appears sufficient.

5.4 Reducing Burstiness

To illustrate the burstiness resulting from improved peering, consider a fully-connected clique of 120 CDN nodes that begin fetching a large file simultaneously. If all have the same peer set, then each node in the replica set k will receive 120/k requests, each for a 60KB chunk. Assuming 2 replicas, the traffic demand on each is 28.8 Mbits. Assuming a 10 Mbps link, it will be fully utilized for 3 seconds just for this chunk, and then the utilization will drop until the next burst of chunks.

The simplest way of reducing the short time-scale node congestion is to increase the number of replicas for each chunk, but this would increase the number of fetches to the origin. Instead, we can improve on the purely mesh-based topology by taking some elements of the stream-oriented systems, which are excellent for reducing link stress. These systems all build communication trees, which eliminates the need to have the same data traverse a link multiple times. While trees are an unattractive option for standard Web CDNs because they add extra latency to every request fetched from the origin, a hybrid scheme can help the large-file case, if the extra hops can reduce congestion.

We take the re-forwarding support to forward misdirected chunks, and use it to create broader routing trees in the peer sets. We change the re-forwarding logic to use a different number of replicas when calculating the HRW set, leading to a broad replica set and a smaller set of nodes that fetch from the origin. We set the *Num-Candidates* value to 1 when evaluating the re-forwarding logic, while dynamically selecting the value at the first proxy. The larger replica set at the first hop reduces the burstiness at any node without increasing origin load.

To dynamically select the number of replicas, we observe that we can eliminate burstiness by spreading the requests equally across the peers at all times. With a target per-client memory consumption, we can determine how many chunks are issued in parallel. So, the replication factor is governed by the following equation:

$$replication factor = \frac{peersize*chunksize}{memory consumption} \quad (1)$$

At 1 MB of buffer space per client, a 60KB chunk size, and 120 peers, our replication factor will be 7. We can, of course, cap the number of peers at some reasonable fraction of the maximum number of peers so that memory pressure does not cause runaway replication for the sake of load balancing. In practice, we limit the replication factor to 20% of the minimum target peer set, which yields a maximum factor of 12.

5.5 Dynamic Window Scaling

Although parallel chunk downloads can exploit multipath bandwidth and reduce the effect of slow transfers, using a fixed number of parallel chunks also has some congestion-related drawbacks which we address. When the content is not cached, the origin server may receive more simultaneous requests than it can handle if each client is using a large number of parallel chunks. For example, the Apache Web Server is configured by default to allow 150 simultaneous connection, and some sites may not have changed this value. If a CDN node has limited bandwidth to the rest of the CDN, too many parallel fetches can cause self-congestion, possibly underutilizing bandwidth, and slowing down the time of all fetches. The problem in this scenario is that too many slow chunks will cause more retries than needed.

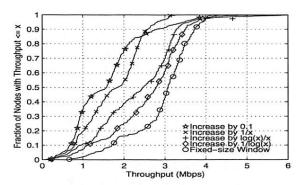


Figure 9: Throughput distribution for various window adjusting functions - Test scheme is described in section 6

In either of these scenarios, using a smaller number of simultaneous fetches would be beneficial, since the perchunk download time would improve. We view finding the "right" number of parallel chunks as a congestion issue, and address it in a manner similar to how TCP performs congestion control. Note that changing the number of parallel chunks is not an attempt to perform low-level TCP congestion control – since the fetches are themselves using TCP, we have this benefit already. Moreover, since the underlying TCP transport is already using additive-increase multiplicative-decrease, we can choose whatever scheme we desire on top of it.

Drawing on TCP Vegas [6], we use the extra information we have in the CoBlitz agent to make the chunk "congestion window" a little more stable than a simple sawtooth. We use three criteria: (1) if the chunk finishes in less than the average time, increase the window, (2) if the first fetch attempt is killed by retries, shrink the window, and (3) otherwise, leave the window size unmodified. We also decide that if more chunk fetches are in progress than the window size dictates, existing fetches are allowed to continue, but no new fetches (including retries) are allowed. Given that our condition for increasing the window is already conservative, we give ourselves some flexibility on exactly how much to add. Similarly, given that the reason for requiring a retry might be that any peer is slow, we decide against using multiplicative decrease when a chunk misses the deadline.

While determining the decrease rate is fairly easy, choosing a reasonable increase rate required some experimentation. The decrease rate was chosen to be one full chunk for each failed chunk, which would have the effect of closing the congestion window very quickly if all of the chunks outstanding were to retry. This logic is less severe than multiplicative decrease if only a small number of chunks miss their deadlines, but can shrink the window to a single chunk within one "RTT" (in this case, average chunk download time) in the case of many

failures.

Some experimentation with different increase rates is shown in Figure 9. The purely additive condition, $\frac{1}{\pi}$ on each fast chunk (where x is the current number of chunks allowed), fares poorly. Even worse is adding onetenth of a chunk per fast chunk, which would be a slow multiplicative increase. The more promising approaches, adding $\frac{log(x)}{x}$ and $\frac{1}{log(x)}$ (where we use log(1) = 1) produce much better results. The $\frac{1}{x}$ case is not surprising, since it will always be no more than additive, since the window grows only when performing well. In TCP, the "slow start" phase would open the window exponentially faster, so we choose to use $\frac{1}{log(x)}$ to achieve a similar effect - it grows relatively quickly at first, and more slowly with larger windows. The chunk congestion window is maintained as a floating-point value, which has a lower bound of 1 chunk, and an upper bound as dictated by the buffer size available, which is normally 60 chunks. The final line in the graph, showing a fixed-size window of 60 chunks, appears to produce better performance, but comes at the cost of a higher node failure rate -2.5 times as many nodes fail to complete with the fixed window size versus the dynamic sizing.

6 Evaluation

In this section, we evaluate the performance of CoBlitz, both in various scenarios, and in comparison with Bit-Torrent [12]. We use BitTorrent because of its wide use in large-file transfer [7], and because other research systems, such as Slurpie, Bullet' and Shark [2, 19, 26], are not running (or in some cases, available) at the time of this writing. As many of these have been evaluated on PlanetLab, we draw some performance and behavior comparisons in Section 7.

One unique aspect of our testing is the scale – we use every running PlanetLab node except those at Princeton, those designated as alpha testing nodes, and those behind firewalls that prevent CoDeeN traffic. The reason for excluding the Princeton nodes is because we place our origin server at Princeton, so the local PlanetLab nodes would exhibit unrealistically large throughputs and skew the means. During our testing in September and early October 2005, the number of available nodes that met the criteria above ranged from 360-380 at any given time, with a union size of 400 nodes.

Our test environment consists of a server with an AMD Sempron processor running at 1.5 GHz, with Linux 2.6.9 as its operating system and lighttpd 1.4.4 [18] as our web server. Our testing consists of downloading a 50MB file in various scenarios. The choice of this file size was to facilitate comparisons with other work [2, 19], which uses file sizes of 40-50MB in their testing. Our testing using a 630MB ISO image for

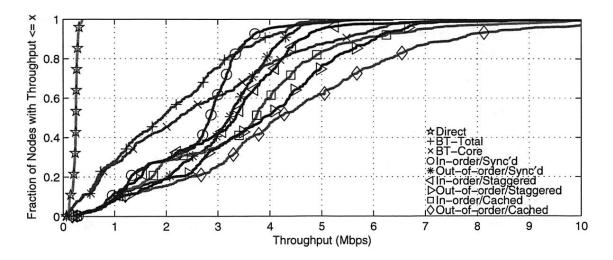


Figure 10: Achieved throughput distribution for all live PlanetLab nodes

the Fedora Core 4 download yielded slightly higher performance, but would complicate comparisons with other systems. Given that some PlanetLab nodes are in parts of the world with limited bandwidth, our use of 50MB files also reduces contention problems for them. Each test is run three times, and the reported numbers are the average value across the tests for which the node was available. Due to the dynamics of PlanetLab, over any long period of time, the set of available nodes will change, and given the span of our testing, this churn is unavoidable.

We tune BitTorrent for performance – the clients and the server are configured to seed the peers indefinitely, and the maximum number of peers is increased to 60. While live BitTorrent usage will have lower performance due to fewer peers and peers departing after downloading, we want the maximum BitTorrent performance.

We test a number of scenarios, as follows:

Direct – all clients fetch from the origin in a single download, which would be typical of standard browsers. For performance, we increase the socket buffer sizes from the system defaults to cover the bandwidth-delay product.

BitTorrent Total – this is a wall-clock timing of Bit-Torrent, which reflects the user's viewpoint. Even when all BitTorrent clients are started simultaneously, downloads begin at different times since clients spend different amounts of time contacting the tracker and finding peers.

BitTorrent Core – this is the BitTorrent performance from the start of the actual downloading at each client. In general, this value is 25-33% higher than the BitTorrent Total time, but is sometimes as much as 4 times larger.

In-order CoBlitz with Synchronization – Clients use CoBlitz to fetch a file for the first time and the chunks are delivered in order. All clients start at the same time.

In-order CoBlitz with Staggering – We stagger the start of each client by the same amount of time that BitTorrent uses before it starts downloading. These stagger times are typically 20 to 40 seconds, with a few outliers as high as 150-230 seconds.

In-order CoBlitz with Contents Cached – Clients ask for a file that has already been fetched previously, and whose chunks are cached at the reverse proxies. All clients begin at the same time.

Out-of-order tests – Out-of-order CoBlitz with Synchronization, Out-of-order CoBlitz with Staggering, and Out-of-order CoBlitz with Contents Cached are the same as their in-order counterparts described above, but the chunks are delivered to the clients out of order.

6.1 Overall Performance

The throughputs and download times for all tests are shown in Figure 10 and Figure 11, with summaries presented in Table 1. For clarity, we trim the x axes of both graphs, and the CDFs shown are of all nodes completing the tests. The actual number of nodes finishing each test are shown in the table. In the throughput graph, lines to the right are more desirable, while in the download time graph, lines to the left are more desirable.

From the graphs, we can see several general trends: all schemes beat direct downloading, uncached CoBlitz generally beats BitTorrent, out-of-order CoBlitz beats

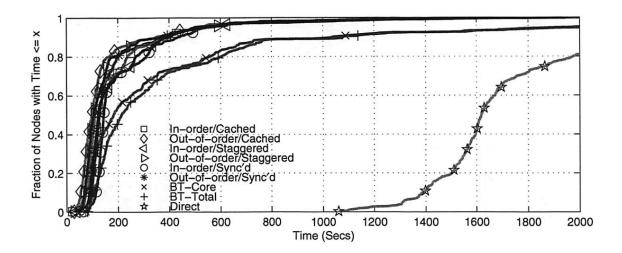


Figure 11: Download times across all live PlanetLab nodes

Strategy	Nodes		Throughput			Download Time		
a E	Good	Failed	Mean	50%	90%	Mean	50%	90%
Direct	372	17-18	0.23	0.20	0.38	1866.8	1618.2	3108.7
BitTorrent-Total	367	21-25	1.97	1.88	3.79	519.0	211.7	1078.3
BitTorrent-Core	367	21-25	2.52	2.19	5.32	485.1	181.1	1036.9
CoBlitz In-order Sync'd	380	8-12	2.50	2.78	3.52	222.4	143.6	434.3
CoBlitz In-order Staggered	383	5-9	2.99	3.26	4.54	122.4	141.7	406.4
CoBlitz In-order Cached	377	12-16	3.51	3.65	5.65	185.2	109.5	389.1
CoBlitz Out-of-order Sync'd	381	8-10	2.91	3.15	4.17	193.9	127.0	381.6
CoBlitz Out-of-order Staggered	384	5-8	3.68	3.78	5.91	105.4	124.6	365.2
CoBlitz Out-of-order Cached	379	8-13	4.36	4.08	7.46	164.3	98.1	349.5

Table 1: Throughputs (in Mbps) and times (in seconds) for various downloading approaches with all live PlanetLab nodes. The count of good nodes is the typical value for nodes completing the download, while the count of failed nodes shows the range of node failures.

in-order delivery, staggered downloading beats synchronized delivery, and cached delivery, even when synchronized, beats the others. Direct downloading at this scale is particularly problematic – we had to abruptly shut down this test because it was consuming most of Princeton's bandwidth and causing noticeable performance degradation.

The worst-case performance for CoBlitz occurs for the uncached case where all clients request the content at exactly the same time and more load is placed on the origin server at once. This case is also very unlikely for regular users, since even a few seconds of difference in start times defeats this problem.

The fairest comparison between BitTorrent and CoBlitz is BT-Total versus CoBlitz out-of-order with Staggering, in which case CoBlitz beats BitTorrent by 55-86% in throughput and factor of 1.7 to 4.94 in download time. Even the worst-case performance for CoBlitz, when all clients are synchronized on uncached content,

generally beats BitTorrent by 27-48% in throughput and a factor of 1.47 to 2.48 in download time.

In assessing how well CoBlitz compares against Bit-Torrent, it is interesting to examine the 90^{th} percentile download times in Table 1 and compare them to the mean and median throughputs. This comparison has appeared in other papers comparing with BitTorrent [19, 26]. We see that the tail of BitTorrent's download times is much worse than comparing the mean or median values. As a result, systems that compare themselves primarily with the worst-case times may be presenting a much more optimistic benefit than seen by the majority of users.

It may be argued that worst-case times are important for systems that need to know an update has been propagated to its members, but if this is an issue, more important than delay is failure to complete. In Table 1, we show the number of nodes that finish each test, and these vary considerably despite the fact that the same set of machines is being used. Of the approximately 400 ma-

	CoBlitz			
Sync	;	Stagg	Stagger	
In-Order Ou	Out	In-Order	Out	
7.0	7.9	9.0	9.0	10.0

Table 2: Bandwidth consumption at the origin, measured in multiples of the file size

chines available across the union of all tests, only about 5-12 nodes fail to complete using CoBlitz, while roughly 17-18 fail in direct testing, and about 21-25 fail with Bit-Torrent. The 5-12 nodes where CoBlitz eventually stops trying to download are at PlanetLab sites with highly-congested links, poor bandwidth, and other problems – India, Australia, and some Switzerland nodes.

6.2 Load at the Origin

Another metric of interest is how much traffic reaches the origin server in these different tests, and this information is provided in Table 2, shown as a multiple of the file size. We see that the CoBlitz scenarios fetch a total of 7 to 9 copies in the various tests, which yields a utility of 43-55 nodes served per fetch (or a cache hit rate of 97.6 - 98.2%). BitTorrent has comparable overall load on the origin, at 10 copies, but has a lower utility value, 35, since it has fewer nodes complete. For Shark, the authors observed it downloading 24 copies from the origin to serve 185 nodes, yielding a utility of 7.7. We believe that part of the difference may stem from peering policy - CoDeeN's unilateral peering approach allows poorlyconnected nodes to benefit from existing clusters, while Coral's latency-oriented clustering may adversely impact the number of fetches needed.

A closer examination of fetches per chunk, shown in Figure 12, shows that CoBlitz's average of 8 copies varies from 4-11 copies by chunk, and these copies appear to be spread fairly evenly geographically. The chunks that receive only 4 fetches are particularly interesting, because they suggest it may be possible to cut CoBlitz's origin load by another factor of 2. We are investigating whether these chunks happen to be served by nodes that overlap with many peer sets, which would further validate CoBlitz's unilateral peering.

6.3 Performance after Flash Crowds

Finally, we evaluate the performance of CoBlitz after a flash crowd, where the CDN nodes can still have the file cached. This was one of motivations for building CoBlitz on top of CoDeeN – that by using an infrastructure geared toward long-duration caching, we could serve the object quickly even after demand for it drops. This test is shown in Figure 13, where clients at all PlanetLab nodes try downloading the file individually, with

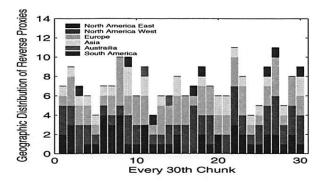


Figure 12: Reverse proxy location distribution

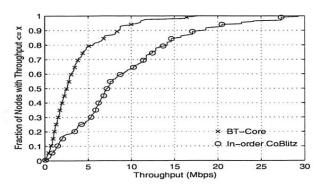
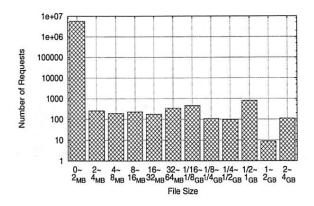


Figure 13: Single node download after flash crowds

no two operating simultaneously. We see that performance is still good after the flash crowd has dissipated – the median for this in-order test is above 7 Mbps, almost tripling the median for in-order uncached and doubling the median of in-order cached. At this bitrate, clients can watch DVD-quality video in real time. We include Bit-Torrent only for comparison purposes, and we see that its median has only marginally improved in this scenario.

6.4 Real-world Usage

One of our main motivations when developing CoBlitz was to build a system that could be used in production, and that could operate with relatively little monitoring. These decisions have led us not only to use simpler, more robust algorithms where possible, but also to restrict the content that we serve. To keep the system usage focused on large-file transfer with a technical focus, and to prevent general-purpose bandwidth cost-shifting, we have placed restrictions on what the general public can serve using CoBlitz. Unless the original file is hosted at a university, CoBlitz will not serve HTML files, most graphics types, and most audio/video formats. As a result of these policies, we have not received any complaints related to the content served by CoBlitz, which has simplified our operational overhead.



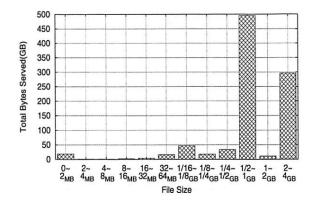


Figure 14: CoBlitz Feb 2006 usage by requests

Figure 15: CoBlitz Feb 2006 usage by bytes served

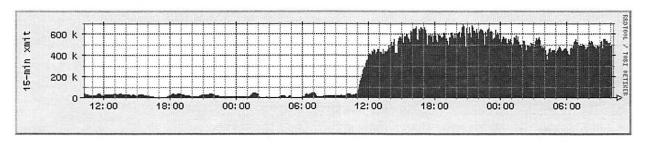


Figure 16: CoBlitz traffic in Kbps on release of Fedora Core 5, averaged over 15-minute intervals. The 5-minute peaks exceeded 700 Mbps.

To get a sense of a typical month's CoBlitz usage, we present the breakdown for February 2006 traffic in Figures 14 (by number of requests) and 15 (by bytes served). Most of the requests for files less than 2MB come from the Stork service [28], which provides package management on PlanetLab, and the CiteSeer Digital Library [11], which provides document downloads via CoBlitz. The two spikes in bytes served are from the Fedora Core Linux distribution, available as either downloadable CD images or DVD images. Most of the remaining traffic comes from smaller sites, other PlanetLab users, and Fedora Core RPM downloads.

A more unusual usage pattern occurred on March 20, 2006, when the Fedora Core 5 Linux distribution was released. Within minutes of the official announcement on the Fedora mailing lists, the availability was mentioned on the front page of Slashdot [27], on a Monday morning for the US. The measurements from this day and the previous day are shown in Figure 16. In less than an hour, CoBlitz went from an average of 20Mbps of traffic to over 400 Mbps, and sustained 5-minute peaks exceeded 700Mbps. CoBlitz functioned as expected, with one exception – many of the clients were using "download agents" that fetch files using a "no-cache" HTTP header. CoBlitz had been honoring these requests for PlanetLab

researchers who wanted to force refreshes, and we had not seen a problem in other environments. However, for this download, these headers were causing unnecessary fetches to the origin that were impacting performance. We made a policy decision to disregard these headers for the Fedora Mirror sites, at which point origin traffic dropped dramatically. This flash crowd had a relatively long tail – it average 200-250Mbps on the third day, and only dropped to less than 100Mbps on the fifth day, a weekend. The memory footprint of CoBlitz was also low – even serving the CD and DVD images on several platforms (PPC, i386, x86_64), the average memory consumption was only 75MB per node.

7 Related Work

Several projects that perform large file transfer have been measured on PlanetLab, with the most closely related ones being Bullet' [19], and Shark [2], which is built on Coral [15]. Though neither system is currently accessible to the public, both have been evaluated recently. Bullet', which operates out-of-order and uses UDP, is reported to achieve 7 Mbps when run on 41 PlanetLab hosts at different sites. In testing under similar conditions, CoBlitz achieves 7.4 Mbps (uncached) and 10.6 Mbps (cached) on average. We could potentially achieve even higher re-

	Shark	CoBlitz				Bullet'
		Uncached		Cached		1
		In	Out	In	Out	1
# Nodes	185	41	41	41	41	41
Median	1.0	6.8	7.4	7.4	9.2	
Mean		7.0	7.4	8.4	10.6	7.0

Table 3: Throughput results (in Mbps) for various systems at specified deployment sizes on PlanetLab. All measurements are for 50MB files, except for Shark, which uses 40MB.

sults by using a UDP-based transport protocol, but our experience suggests that UDP traffic causes more problems, both from intrusion detection systems as well as stateful firewalls. Shark's performance for transferring a 40MB file across 185 PlanetLab nodes shows a median throughput of 0.96 Mbps. As discussed earlier, Shark serves an average of only 7.7 nodes per fetch, which suggests that their performance may improve if they use techniques similar to ours to reduce origin server load. The results for all of these systems are shown in Table 3. The missing data for Bullet' and Shark reflect the lack of information in the publications, or difficulty extracting the data from the provided graphs.

The use of parallel downloads to fetch a file has been explored before, but in a more narrow context - Rodriguez et al. use HTTP byte-range queries to simultaneously download chunks in parallel from different mirror sites [23]. Their primary goal was to improve single client downloading performance, and the full file is pre-populated on all of their mirrors. What distinguishes CoBlitz from this earlier work is that we make no assumptions about the existence of the file on peers, and we focus on maintaining stability of the system even when a large number of nodes are trying to download simultaneously. CoBlitz works if the chunks are fully cached, partially cached, or not at all cached, fetching any missing chunks from the origin as needed. In the event that many chunks need to be fetched from the origin, CoBlitz attempts to reduce origin server overload. Finally, from a performance standpoint, CoBlitz attempts to optimize the memory cache hit rate for chunks, something not considered in Rodriguez's system.

While comparing with other work is difficult due to the difference in test environment, we can make some informed conjecture based on our experiences. Fast-Replica's evaluation includes tests of 4-8 clients, and their per-client throughput drops from 5.5 Mbps with 4 clients to 3.6 Mbps with 8 clients [9]. Given that their file is broken into a small number of equal-sized pieces, the slowest node in the system is the overall bottleneck. By using a large number of small, fixed-size pieces, CoBlitz can mitigate the effects of slow nodes, either by increas-

ing the number of parallel fetches, or by retrying chunks that are too slow. Another system, Slurpie, limits the number of clients that can access the system at once by having each one randomly back off such that only a small number are contacting the server regardless of the number of nodes that want service. Their local-area testing has clients contact the server at the rate of one every three seconds, which staggers it far more than BitTorrent. Slurpie's evaluation on PlanetLab provides no absolute performance numbers [26], making it difficult to draw comparisons. However, their performance appears to degrade beyond 16 nodes.

The scarcity of deployed systems for head-to-head comparisons supports part of our motivation – by reusing CDN infrastructure, we have been able to easily deploy CoBlitz and keep it running.

8 Conclusion

We show that, with a relatively small amount of modification, a traditional, HTTP-based content distribution network can be made to efficiently support scalable large-file transfer. Even with no modifications to clients, servers, or client-side software, our approach provides good performance under demanding conditions, but can provide even higher performance if clients implement a relatively simple HTTP feature, chunked encoding.

Additionally, we show how we have taken the experience gained from 18 months of CoBlitz deployment, and used it to adapt our algorithms to be more aware of real-world conditions. We demonstrate the advantages provided by this approach by evaluating CoBlitz's performance across all of PlanetLab, where it exceeds the performance of BitTorrent as well as all other research efforts known to us.

In the process of making CoBlitz handle scale and reduce congestion both within the CDN and at the origin server, we identify a number of techniques and observations that we believe can be applied to other systems of this type. Among them are: (a) using unilateral peering, which simplifies communication as well as enabling the inclusion of policy-limited or poorly-connected nodes, (b) using request re-forwarding to reduce the origin server load when nodes send requests to an overly-broad replica set, (c) dynamically adjusting replica sets to reduce burstiness in short time scales, (d) congestion-controlled parallel chunk fetching, to reduce both origin server load as well as self-interference at slower CDN nodes.

We believe that the lessons we have learned from CoBlitz should help not only the designers of future systems, but also provide a better understanding of how to design these kinds of algorithms to reflect the unpredictable behavior we have seen in real deployment.

Acknowledgments

We would like to thank our shepherd, Neil Spring, and the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

- Akamai Technologies Inc., 1999. http://www.akamai.com/.
- [2] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05), Boston, MA, May 2005.
- [3] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany, S. Vadhiyar, and R. Wolski. Logistical computing and internetworking: Middleware for the use of storage in communication. In 3rd Annual International Workshop on Active Middleware Services (AMS), San Francisco, August 2001.
- [4] S. Birrer, D. Lu, F. E. Bustamante, Y. Qiao, and P. Dinda. FatNemo: Building a resilient multi-source multicast fat-tree. In Proceedings of 9th International Workshop on Web Content Caching and Distribution (IWCW'04), Beijing, China, October 2004.
- [5] B. Biskeborn, M. Golightly, K. Park, and V. S. Pai. (Re)Design considerations for scalable large-file content distribution. In Proceedings of Second Workshop on Real, Large Distributed Systems(WORLDS), San Francisco, CA, December 2005.
- [6] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM '94*, August 1994.
- [7] CacheLogic, 2004. http://www.cachelogic.com/news/pr040715.php.
- [8] M. Castro, P. Drushcel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of SOSP'03*, Oct 2003
- [9] L. Cherkasova and J. Lee. FastReplica: Efficient large file distribution within content delivery networks. In *Proceedings of the* 4th USITS, Seattle, WA, March 2003.
- [10] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. In *IEEE Journal on Selected Areas in Communica*tion (JSAC), Special Issue on Networking Support for Multicast, 2002.
- [11] CiteSeer Scientific Literature Digital Library. http://citeseer.ist.psu.edu/.
- [12] B. Cohen. Bittorrent, 2003. http://bitconjurer.org/BitTorrent.
- [13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of SIG-COMM '04*, Portland, Oregon, August 2004.
- [14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.
- [15] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI'04)*, 2004.
- [16] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the 8th Inter*national World-Wide Web Conference, 1999.
- [17] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees:

- Distributed caching protocols for relieving hot spots on the world wide web. In ACM Symposium on Theory of Computing, 1997.
- [18] J. Kneschke. lighttpd. http://www.lightttpd.net.
- [19] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of USENIX Annual Technical Conference*, 2005.
- [20] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In Proceedings of 19th ACM SOSP, 2003.
- [21] B. Maggs. Personal communication (email) with Vivek S. Pai, October 20th, 2005.
- [22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Tech*nical Conference, June 2004.
- [23] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the Internet. In *Proceedings of IEEE Infocom*, Tel-Aviv. Israel. March 2000.
- [24] A. Rousskov. Range requests squid mailing list. http://www.squid-cache.org/mail-archive/ squid-dev/199801/0005.html.
- [25] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. http://www.measurement-factory.com/results/.
- [26] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of IEEE Infocom*, Hong Kong, 2004.
- [27] Slashdot.
 http://slashdot.org/.
- [28] Stork on PlanetLab. http://www.cs.arizona.edu/stork/.
- [29] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.
- [30] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. In ACM Transactions on Computer Systems, May 2003.
- [31] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation(OSDI), Boston, MA, December 2002.
- [32] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In Proceedings of the USENIX Annual Technical Conference, 2004.

Efficient Replica Maintenance for Distributed Storage Systems

Byung-Gon Chun,[†] Frank Dabek,^{*} Andreas Haeberlen,[‡] Emil Sit,^{*} Hakim Weatherspoon,[†] M. Frans Kaashoek,^{*} John Kubiatowicz,[†] and Robert Morris^{*}

* MIT Computer Science and Artificial Intelligence Laboratory,

Abstract

This paper considers replication strategies for storage systems that aggregate the disks of many nodes spread over the Internet. Maintaining replication in such systems can be prohibitively expensive, since every transient network or host failure could potentially lead to copying a server's worth of data over the Internet to maintain replication levels.

The following insights in designing an efficient replication algorithm emerge from the paper's analysis. First, durability can be provided separately from availability; the former is less expensive to ensure and a more useful goal for many wide-area applications. Second, the focus of a durability algorithm must be to create new copies of data objects faster than permanent disk failures destroy the objects; careful choice of policies for what nodes should hold what data can decrease repair time. Third, increasing the number of replicas of each data object does not help a system tolerate a higher disk failure probability, but does help tolerate bursts of failures. Finally, ensuring that the system makes use of replicas that recover after temporary failure is critical to efficiency.

Based on these insights, the paper proposes the Carbonite replication algorithm for keeping data durable at a low cost. A simulation of Carbonite storing 1 TB of data over a 365 day trace of PlanetLab activity shows that Carbonite is able to keep all data durable and uses 44% more network traffic than a hypothetical system that only responds to permanent failures. In comparison, Total Recall and DHash require almost a factor of two more network traffic than this hypothetical system.

1 Introduction

Wide-area distributed storage systems typically use replication to provide two related properties: durability and availability. *Durability* means that objects that an application has put into the system are not lost due to disk failure whereas *availability* means that get will be able to return the object promptly. Objects can be durably stored but not

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, http://project-iris.net/. Andreas Haeberlen was supported in part by the Max Planck Society. Emil Sit was supported in part by the Cambridge-MIT Institute. Hakim Weatherspoon was supported by an Intel Foundation PhD Fellowship.

immediately available: if the only copy of an object is on the disk of a node that is currently powered off, but will someday re-join the system with disk contents intact, then that object is durable but not currently available. The paper's goal is to develop an algorithm to store immutable objects durably and at a low bandwidth cost in a system that aggregates the disks of many Internet nodes.

The threat to durability is losing the last copy of an object due to permanent failures of disks. Efficiently countering this threat to durability involves three main challenges. First, network bandwidth is a scarce resource in a wide-area distributed storage system. To store objects durably, there must be enough network capacity to create copies of objects faster than they are lost due to disk failure. Second, a system cannot always distinguish between transient failures and permanent disk failures: it may waste network bandwidth by creating new copies during transient failures. Third, after recovery from transient failures, some replicas may be on nodes that the replica lookup algorithm does not query and are thus effectively lost.

Since transient failures are common in wide-area systems, replication algorithms can waste bandwidth by making unneeded replicas. For example, the initial replication algorithm [6] that the DHash distributed hash table (DHT) [9] turned out to be inadequate to build storage applications such as UsenetDHT [34], Antiquity [11], and OverCite [35, 36].

A problem with DHash was that its design was driven by the goal of achieving 100% availability; this decision caused it to waste bandwidth by creating new replicas in response to temporary failures. Its design and similar ones (such as Total Recall [3]) are overkill for durability. Furthermore, users of many Internet applications can tolerate some unavailability. For example, Usenet readers will see all articles eventually, as long as they are stored durably. Our experience with these DHT applications has led us to the following insights:

• Durability is a more practical and useful goal than availability for applications that store objects (as op-

[‡] Rice University/MPI-SWS, † University of California, Berkeley

posed to caching objects).

- The main goal of a durability algorithm should be to create new copies of an object faster than they are destroyed by disk failures; the choice of how replicas are distributed among nodes can make this task easier.
- Increasing the replication level does not help tolerate a higher average permanent failure rate, but it does help cope with bursts of failures.
- Reintegrating returning replicas is key to avoiding unnecessary copying.

Using these insights we have developed Carbonite, an efficient wide-area replication algorithm for keeping objects durable. After inserting a set of initial replicas, Carbonite begins by creating new replicas mostly in response to transient failures. However, over time it is increasingly able to ignore transient failures and approaches the goal of only producing replicas in response to permanent failures.

Carbonite's design assumes that the disks in the distributed storage system fail independently of each other: failures of geographically distributed hard drives from different manufacturers are likely to be uncorrelated.

In a year-long PlanetLab failure trace, however, we observe some correlated failures because of coordinated reinstalls of the PlanetLab software. Despite this, an evaluation using the PlanetLab failure trace shows that Carbonite is able to keep 1 TB of data durable, and consumes only 44% more network traffic than a hypothetical system that only responds to permanent failures. In comparison, Total Recall and DHash require almost a factor of two more network traffic than this hypothetical system.

The rest of this paper explains our durability models and algorithms, interleaving evaluation results into the explanation. Section 2 describes the simulated evaluation environment. Section 3 presents a model of the relationship between network capacity, amount of replicated data, number of replicas, and durability. Section 4 explains how to decrease repair time, and thus increase durability, by proper placement of replicas on servers. Section 5 presents an algorithm that reduces the bandwidth wasted making copies due to transient failures. Section 6 outlines some of the challenges that face practical implementations of these ideas, Section 7 discusses related work, and Section 8 concludes.

2 System environment

The behavior of a replication algorithm depends on the environment in which it is used: high disk failure rates or low network access link speeds make it difficult for any system to maintain durability. We will use the characteristics of the PlanetLab testbed as a representative environment when evaluating wide-area replication techniques.

Dates 1 March 2005 – 28 Feb 2006

Number of hosts 632

Number of transient failures 21255

Number of disk failures 219

Transient host downtime (s) 1208, 104647, 14242

Any failure interarrival (s) 305, 1467, 3306

Disk failures interarrival (s) 54411, 143476, 490047

Table 1: CoMon+PLC trace characteristics.

For explanatory purposes, we will also use a synthetic trace that makes some of the underlying trends more visible. This section describes both environments, as well as the simulator we used to evaluate our algorithm.

2.1 PlanetLab characteristics

(Median/Mean/90th percentile)

PlanetLab is a large (> 600 node) research testbed [28] with nodes located around the world. We chose this testbed as our representative environment mainly because it is a large, distributed collection of machines that has been monitored for long periods; we use this monitoring data to construct a realistic trace of failures in a mostly managed environment.

The main characteristics of PlanetLab that interest us are the rates of disk and transient failures. We use historical data collected by the CoMon project [25] to identify transient failures. CoMon has archival records collected on average every 5 minutes that include the uptime as reported by the system uptime counter on each node. We use resets of this counter to detect reboots, and we estimate the time when the node became unreachable based on the last time CoMon was able to successfully contact the node. This allows us to pinpoint failures without depending on the reachability of the node from the CoMon monitoring site.

We define a disk failure to be any permanent loss of disk contents, due to disk hardware failure or because its contents are erased accidentally or intentionally. In order to identify disk failures, the CoMon measurements were supplemented with event logs from PlanetLab Central [28]. This database automatically records each time a PlanetLab node is reinstalled (e.g., for an upgrade, or after a disk is replaced following a failure). The machine is then considered offline until the machine is assigned a regular boot state in the database. Table 1 summarizes the statistics of this trace. Figure 7(a) visualizes how transient and disk failures accumulate over time in this network.

2.2 Synthetic trace

We also generated synthetic traces of failures by drawing failure inter-arrival times from exponential distributions. Synthetic traces have two benefits. First, they let us simulate longer time periods, and second, they allow us to increase the failure density, which makes the basic underlying trends more visible. We conjecture that exponential inter-failure times are a good model for disks that are independently acquired and operated at geographically separated sites; exponential intervals are possibly not so well justified for transient failures due to network problems.

Each synthetic trace contains 632 nodes, just like the PlanetLab trace. The mean session time and downtime match the values shown in Table 1; however, in order to increase the failure density, we extended the length to two years and reduced the average node lifetime to one year. Each experiment was run with ten different traces; the figures show the averages from these experiments.

2.3 Simulation

We use the failure traces to drive an event-based simulator. In the simulator, each node has unlimited disk capacity, but limited link bandwidth. However, it assumes that all network paths are independent so that there are no shared bottlenecks. Further it assumes that if a node is available, it is reachable from all other nodes. This is occasionally not the case on PlanetLab [14]; however, techniques do exist to mask the effects of partially unreachable nodes [1].

The simulator takes as input a trace of transient and disk failure events, node repairs and object insertions. It simulates the behavior of nodes under different protocols and produces a trace of the availability of objects and the amount of data sent and stored by each node for each hour of simulated time. Each simulation calls put with 50,000 data objects, each of size 20 MB. Unless otherwise noted, each node is configured with an access link capacity of 150 KBytes/s, roughly corresponding to the throughput achievable under the bandwidth cap imposed by Planet-Lab. The goal of the simulations is to show the percentage of objects lost and the amount of bandwidth needed to sustain objects over time.

3 Understanding durability

We consider the problem of providing durability for a storage system composed of a large number of nodes spread over the Internet, each contributing disk space. The system stores a large number of independent pieces of data. Each piece of data is immutable. The system must have a way to name and locate data; the former is beyond the scope of this work, while the latter may affect the possible policies for placing replicas. While parts of the system will suffer temporary failures, such as network partitions or power failures, the focus of this section is on failures that result in permanent loss of data. Section 5 shows how to efficiently manage transient failures; this section describes some fundamental constraints and challenges in providing durability.

3.1 Challenges to durability

It is useful to view permanent disk and node failures as having an average rate and a degree of burstiness. To provide high durability, a system must be able to cope with both.

In order to handle some average rate of failure, a highdurability system must have the ability to create new replicas of objects faster than replicas are destroyed. Whether the system can do so depends on the per-node network access link speed, the number of nodes (and hence access links) that help perform each repair, and the amount of data stored on each failed node. When a node n fails, the other nodes holding replicas of the objects stored on nmust generate replacements: objects will remain durable if there is sufficient bandwidth available on average for the lost replicas to be recreated. For example, in a symmetric system each node must have sufficient bandwidth to copy the equivalent of all data it stores to other nodes during its lifetime.

If nodes are unable to keep pace with the average failure rate, no replication policy can prevent objects from being lost. These systems are *infeasible*. If the system is infeasible, it will eventually "adapt" to the failure rate by discarding objects until it becomes feasible to store the remaining amount of data. A system designer may not have control over access link speeds and the amount of data to be stored; fortunately, choice of object placement can improve the speed that a system can create new replicas as discussed in Section 4.

If the creation rate is only slightly above the average failure rate, then a burst of failures may destroy all of an object's replicas before a new replica can be made; a subsequent lull in failures below the average rate will not help replace replicas if no replicas remain. For our purposes, these failures are *simultaneous*: they occur closer together in time than the time required to create new replicas of the data that was stored on the failed disk. Simultaneous failures pose a constraint tighter than just meeting the average failure rate: every object must have more replicas than the largest expected burst of failures. We study systems that aim to maintain a target number of replicas in order to survive bursts of failure; we call this target r_L .

Higher values of r_L do *not* allow the system to survive a higher average failure rate. For examples, if failures were to arrive at fixed intervals, then either $r_L = 2$ would always be sufficient, or no amount of replication would ensure durability. If $r_L = 2$ is sufficient, there will always be time to create a new replica of the objects on the most recently failed disk before their remaining replicas fail. If creating new replicas takes longer than the average time between failures, no fixed replication level will make the system feasible; setting a replication level higher than two would only increase the number of bytes each node must copy in response to failures, which is already infeasible at $r_L = 2$.

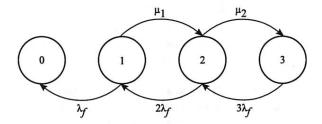


Figure 1: A continuous time Markov model for the process of replica failure and repair for a system that maintains three replicas ($r_L = 3$). Numbered states correspond to the number of replicas of each object that are durable. Transitions to the left occur at the rate at which replicas are lost; right-moving transitions happen at the replica creation rate.

3.2 Creation versus failure rate

It might seem that any creation rate higher than the average failure rate will lead to an unbounded number of replicas, thus satisfying the burst constraint. However, this intuition is false. To see why, let us model the number of replicas of an object as a birth-death process using a continuous time Markov chain, which assumes independent exponential inter-failure and inter-repair times. This assumption is reasonable for independent disk failures.

An object is in state i when i disks hold a replica of the object. There are thus r_L+1 possible states, as we start with r_L replicas and only create new replicas in response to failures. From a given state i, there is a transition to state i+1 with rate μ_i corresponding to repair, except for state 0 which corresponds to loss of durability and state r_L which does not need repair. The actual rate μ_i depends on how bandwidth is allocated to repair and may change depending on the replication level of an object. There is a transition to the next lower state i-1 with rate $i\lambda_f$ because each of the i nodes holding an existing replica might fail. Figure 1 shows this model for the case where $r_L=3$.

This model can be analyzed numerically to shed light on the impact of r_L on the probability of data loss; we will show this in Section 3.3. However, to gain some intuition about the relationship between creation and failure rates and the impact this has on the number of replicas that can be supported, we consider a simplification of Figure 1 that uses a fixed μ but repairs constantly, even allowing for transitions out of state 0. While these changes make the model less realistic, they turn the model into an $M/M/\infty$ queue [19] where the "arrival rate" is the repair rate and the "service rate" is the per-replica failure rate. The "number of busy servers" is the number of replicas: the more replicas an object has, the more probable it is that one of them will fail.

This simplification allows us to estimate the equilibrium number of replicas: it is μ/λ_f . Given μ and λ_f , a

system cannot expect to support more than this number of replicas. For example, if the system must handle coincidental bursts of five failures, it must be able to support at least six replicas and hence the replica creation rate must be at least 6 times higher than the average replica failure rate. We will refer to μ/λ_f as θ . Choices for r_L are effectively limited by θ . It is not the case that durability increases continuously with r_L ; rather, when using $r_L > \theta$, the system provides the best durability it can, given its resource constraints. Higher values of θ decrease the time it takes to repair an object, and thus the 'window of vulnerability' during which additional failures can cause the object to be destroyed.

To get an idea of a real-world value of θ , we estimate μ and λ_f from the historical failure record for disks on PlanetLab. From Table 1, the average disk failure inter-arrival time for the entire test bed is 39.85 hours. On average, there were 490 nodes in the system, so we can estimate the mean time between failures for a single disk as 490 · 39.85 hours or 2.23 years. This translates to $\lambda_f \approx 0.439$ disk failures per year.

The replica creation rate μ depends on the achievable network throughput per node, as well as the amount of data that each node has to store (including replication). PlanetLab currently limits the available network bandwidth to 150 KB/s per node, and if we assume that the system stores 500 GB of unique data per node with $r_L=3$ replicas each, then each of the 490 nodes stores 1.5 TB. This means that one node's data can be recreated in 121 days, or approximately three times per year. This yields $\mu \approx 3$ disk copies per year.

In a system with these characteristics, we can estimate $\theta = \mu/\lambda_f \approx 6.85$, though the actual value is likely to be lower. Note that this ratio represents the equilibrium number of *disks* worth of data that can be supported; if a disk is lost, all replicas on that disk are lost. When viewed in terms of disk failures and copies, θ depends on the value of r_L : as r_L increases, the total amount of data stored per disk (assuming available capacity) increases proportionally and reduces μ . If $\lambda_f = \mu$, the system can in fact maintain r_L replicas of each object.

To show the impact of θ , we ran an experiment with the synthetic trace (i.e., with 632 nodes, a failure rate of $\lambda_f = 1$ per year and a storage load of 1 TB), varying the available bandwidth per node. In this case, 100 B/s corresponds to $\theta = 1.81/r_L$. Figure 2 shows that, as θ drops below one, the system can no longer maintain full replication and starts operating in a 'best effort' mode, where higher values of r_L do not give any benefit. The exception is if some of the initial r_L replicas survive through the entire trace, which explains the small differences on the left side of the graph.

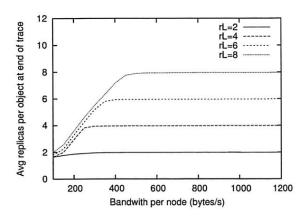


Figure 2: Average number of replicas per object at the end of a two-year synthetic trace for varying values of θ , which varies with bandwidth per node (on the *x*-axis) and total data stored (r_L) . Where $\theta < 1$, the system cannot maintain the full replication level; increasing r_L further does not have any effect.

3.3 Choosing r_L

A system designer must choose an appropriate value of r_L to meet a target level of durability. That is, for a given deployment environment, r_L must be high enough so that a burst of r_L failures is sufficiently rare.

One approach is to set r_L to one more than the maximum burst of simultaneous failures in a trace of a real system. For example, Figure 3 shows the burstiness of permanent failures in the PlanetLab trace by counting the number of times that a given number of failures occurs in disjoint 24 hour and 72 hour periods. If the size of a failure burst exceeds the number of replicas, some objects may be lost. From this, one might conclude that 12 replicas are needed to maintain the desired durability. This value would likely provide durability but at a high cost. If a lower value of r_L would suffice, the bandwidth spent maintaining the extra replicas would be wasted.

There are several factors to consider in choosing r_L to provide a certain level of durability. First, even if failures are independent, there is a non-zero (though small) probability for every burst size up to the total number of nodes. Second, a burst may arrive while there are fewer than r_L replicas. One could conclude from these properties that the highest possible value of r_L is desirable. On the other hand, the simultaneous failure of even a large fraction of nodes may not destroy any objects, depending on how the system places replicas (see Section 4). Also, the workload may change over time, affecting μ and thus θ .

The continuous time Markov model described in Figure 1 reflects the distributions of both burst size and object replication level. The effect of these distributions is signif-

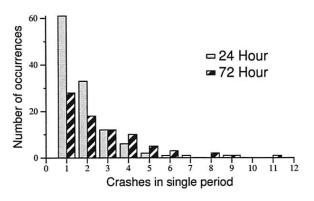


Figure 3: Frequency of "simultaneous" failures in the PlanetLab trace. These counts are derived from breaking the trace into non-overlapping 24 and 72 hour periods and noting the number of permanent failures that occur in each period. If there are x replicas of an object, there were y chances in the trace for the object to be lost; this would happen if the remaining replicas were not able to respond quickly enough to create new replicas of the object.

icant. An analysis of the governing differential equations can be used to derive the probability that an object will be at a given replication level after a given amount of time. In particular, we can determine the probability that the chain is in state 0, corresponding to a loss of durability.

We show the results of such an analysis in Figure 4; for details, see [7]. To explore different workloads, we consider different amounts of data per node. The graph shows the probability that an object will survive after four years as a function of r_L and data stored per node (which affects the repair rate and hence θ).

As r_L increases, the system can tolerate more simultaneous failures and objects are more likely to survive. The probability of object loss at $r_L = 1$ corresponds to using no replication. This value is the same for all curves since it depends only on the lifetime of a disk; no new replicas can be created once the only replica of the object is lost. To store 50 GB durably, the system must use an r_L of at least 3. As the total amount of data increases, the r_L required to attain a given survival probability also increases. Experiments confirm that data is lost on the PlanetLab trace only when maintaining fewer than three replicas.

4 Improving repair time

This section explores how the system can increase durability by replacing replicas from a failed disk in parallel. In effect, this reduces the time needed to repair the disk and increases θ .

Each node, *n*, designates a set of other nodes that can potentially hold copies of the objects that *n* is responsible for. We will call the size of that set the node's *scope*, and

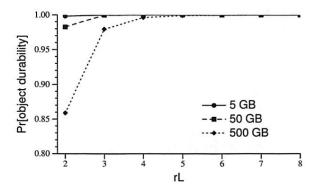


Figure 4: Analytic prediction for object durability after four years on PlanetLab. The x-axis shows the initial number of replicas for each object: as the number of replicas is increased, object durability also increases. Each curve plots a different per-node storage load; as load increases, it takes longer to copy objects after a failure and it is more likely that objects will be lost due to simultaneous failures.

consider only system designs in which every node has the same scope. Scope can range from a minimum of r_L to a maximum of the number of nodes in the system.

A small scope means that all the objects stored on node n have copies on nodes chosen from the same restricted set of other nodes. The advantage of a small scope is that it makes it easier to keep track of the copies of each object. For example, DHash stores the copies of all the objects with keys in a particular range on the successor nodes of that key range; the result is that those nodes store similar sets of objects, and can exchange compressed summaries of the objects they store when they want to check that each object is replicated a sufficient number of times [6].

The disadvantage of a small scope is that the effort of creating new copies of objects stored on a failed disk falls on the small set of nodes in that disk's scope. The time required to create the new copies is proportional to the amount of data on one disk divided by the scope. Thus a small scope results in a long recovery time. Another problem with a small scope, when coupled with consistent hashing, is that the addition of a new node may cause needless copying of objects: the small scope may dictate that the new node replicate certain objects, forcing the previous replicas out of scope and thus preventing them from contributing to durability.

Larger scopes spread the work of making new copies of objects on a failed disk over more access links, so that the copying can be completed faster. In the extreme of a scope of N (the number of nodes in the system), the remaining copies of the objects on a failed disk would be spread over all nodes, assuming that there are many more objects than nodes. Furthermore, the new object copies created after the failure would also be spread over all the

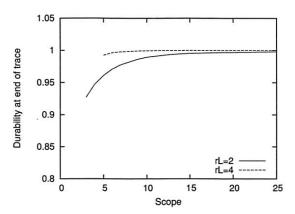


Figure 5: Durability for different scopes in a synthetic trace with low θ . Larger scopes spread the repair work over more access links and improve the nodes' ability to monitor replicas and temporary failures, which results in higher durability.

nodes. Thus the network traffic sources and destinations are spread over all the access links, and the time to recover from the failure is short (proportional to the amount of data on one disk divided by N).

A larger scope also means that a temporary failure will be noticed by a larger number of nodes. Thus, more access links are available to create additional replicas while the failure lasts. Unless these links are already fully utilized, this increases the effective replica creation rate, and thus improves durability.

Figure 5 shows how scope (and thus repair time) affects object durability in a simulation on a synthetic trace. To reduce θ , we limit the bandwidth per node to 1000 B/s in this experiment. We vary the repair threshold and the scope, and measure durability after two years of simulated time. Increasing the scope from 5 to 25 nodes reduces the fraction of lost objects by an order of magnitude, independent of r_L . By including more nodes (and thus more network connections) in each repair effort, the work is spread over more access links and completes faster, limiting the window of time in which the system is vulnerable to another disk failure. Ideally, by doubling the scope, the window of vulnerability can be cut in half.

A large scope reduces repair time and increases durability; however, implementing a large scope presents two trade-offs. First, the system must monitor each node in the scope to determine the replication levels; when using a large scope, the system must monitor many nodes. This increased monitoring traffic limits scalability. Second, in some instances, a large scope can increase the likelihood that a simultaneous failure of multiple disks will cause some object to be lost.

If objects are placed randomly with scope N and there are many objects, then it is likely that all $\binom{N}{r_L}$ potential

replica sets are used. In this scenario, the simultaneous failure of any r_L disks is likely to cause data loss: there is likely to be at least one object replicated on exactly those disks. A small scope limits placement possibilities that are used, concentrating objects into common replica sets. As a result, it is less likely that a given set of r_L failures will affect a replica set, but when data loss does occur, many more objects will be lost. These effects exactly balance: the expected number of objects lost during a large failure event is identical for both strategies. It is the variance that differs between the two strategies.

5 Reducing transient costs

The possibility of transient failures complicates providing durability efficiently: we do not want to make new copies in response to transient failures, but it is impossible to distinguish between disk failures and transient failures using only remote network measurements. This section focuses minimizing the amount of network traffic sent in response to transient failures.

The key technique needed to achieve this is to ensure that the system reintegrates object replicas stored on nodes after transient failures; this means the system must be able to track more than r_L replicas of each object. The number of replicas that the system must remember turns out to be dependent on a, the average fraction of time that a node is available. However, we show that the correct number of extra replicas can be determined without estimating a by tracking the location of all replicas, including those that are offline. We introduce the Carbonite algorithm that uses this technique and demonstrate its effectiveness using simulations.

We additionally consider two other techniques for limiting response to transient failures: creating extra replicas in batches and using timeouts as a heuristic for distinguishing transient from disk failures. Both are of limited value: batching is best able to save bandwidth when using erasure codes and, in the presence of reintegration, timeouts work well only if node downtimes are notably shorter than node (and disk) lifetimes.

5.1 Carbonite details

The Carbonite maintenance algorithm focuses on reintegration to avoid responding to transient failures. Durability is provided by selecting a suitable value of r_L ; an implementation of Carbonite should place objects to maximize θ and preferentially repair the least replicated object. Within these settings, Carbonite works to efficiently maintain r_L copies, thus providing durability.

Because it is not possible to distinguish between transient and disk failures remotely, Carbonite simply responds to any detected failure by creating a new replica. This approach is shown in Figure 6. If fewer than r_L replicas are detected as available, the algorithm creates enough

// Iterate through the object database
// and schedule an object for repair if needed
MAINTAIN_REPLICAS()
keys = <DB.object_keys sorted number of available replicas>
foreach k in keys:
 n = replicas[k].len ()
 if (n < r_L)
 newreplica = enqueue_repair (k)
 replicas[k].append (newreplica)</pre>

Figure 6: Each node maintains a list of objects for which it is responsible and monitors the replication level of each object using some synchronization mechanism. In this code, this state is stored in the replicas hash table though an implementation may choose to store it on disk. This code is called periodically to enqueue repairs on those objects that have too few replicas available; the application can issue these requests at its convenience.

new replicas to return the replication level to r_L .

However, Carbonite remembers which replicas were stored on nodes that have failed so that they can be reused if they return. This allows Carbonite to greatly reduce the cost of responding to transient failures. For example, if the system has created two replicas beyond r_L and both fail, no work needs to be done unless a third replica fails before one of the two currently unavailable replicas returns. Once enough extra replicas have been created, it is unlikely that fewer than r_L of them will be available at any given time. Over time, it is increasingly unlikely that the system will need to make any more replicas.

5.2 Reintegration reduces maintenance

Figure 7 shows the importance of reintegrating replicas back into the system by comparing the behavior of Carbonite to two prior DHT systems and a hypothetical system that can differentiate disk from transient failures using an oracle and thus only reacts to disk failures. In the simulation, each system operates with $r_L = 3$. The systems are simulated against the PlanetLab trace (a) and a synthetic trace (b). The *y*-axes plot the cumulative number of bytes of network traffic used to create replicas; the *x*-axes show time.

Unlike all other synthetic traces used in this paper, whose parameters are different from the PlanetLab trace in order to bring out the basic underlying trends, the synthetic trace used in Figure 7 was configured to be similar to the PlanetLab trace. In particular, the average node lifetime and the median downtime are the same. The result is still an approximation (for example, PlanetLab grew during the trace) but the observed performance is similar. Some of the observed differences are due to batching (used by Total Recall) and timeouts (used by all systems); the impact of these are discussed in more detail in Sections 5.4 and 5.5.

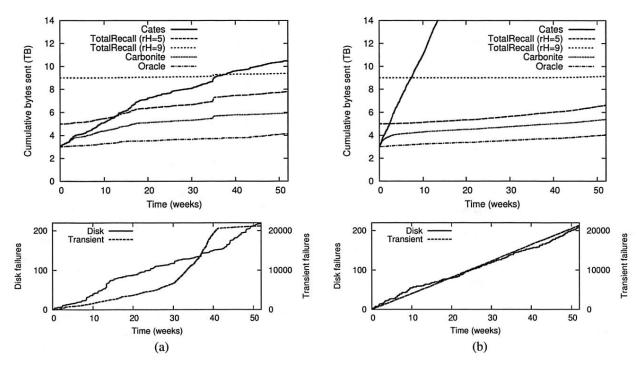


Figure 7: A comparison of the total amount of work done by different maintenance algorithms with $r_L = 3$ using a PlanetLab trace (left) and a synthetic trace (right). In all cases, no objects are lost. However, $r_L = 2$ is insufficient: for the PlanetLab trace, even a system that could distinguish permanent from transient failures would lose several objects.

Since the oracle system responds only to disk failures, it uses the lowest amount of bandwidth. The line labeled Cates shows a system that keeps track of exactly r_L replicas per object; this system approximates the behavior of DHTs like DHash, PAST and OpenDHT. Each failure causes the number of replicas to drop below r_L and causes this system to create a new copy of an object, even if the failure was transient. If the replica comes back online, it is discarded. This behavior results in the highest traffic rate shown. The difference in performance between the PlanetLab and Poisson trace is due to differences in the distribution of downtimes: Poisson is not a particularly good fit for the PlanetLab downtime distribution.

Total Recall [3] tracks up to a fixed number of replicas, controlled by a parameter r_H ; we show $r_H = 5$ which is optimal for these traces, and $r_H = 9$. As can be seen at the right of the graphs, this tracking of additional replicas allows Total Recall to create fewer replicas than the Cates system. When more than r_L replicas are available, a transient failure will not cause Total Recall to make a new copy. However, Total Recall's performance is very sensitive to r_H . If r_H is set too low, a series of transient failures will cause the replication level to drop below r_L and force it to create an unnecessary copy. This will cause Total Recall to approach Cates (when $r_H = r_L$). Worse, when the system creates new copies it forgets about any copies that are currently on failed nodes and cannot benefit from the

return of those copies. Without a sufficiently long memory, Total Recall must make additional replicas. Setting r_H too high imposes a very high insertion cost and results in work that may not be needed for a long time.

Carbonite reintegrates all returning replicas into the replica sets and therefore creates fewer copies than Total Recall. However, Carbonite's inability to distinguish between transient and disk failures means that it produces and maintains more copies than the oracle based algorithm. This is mainly visible in the first weeks of the trace as Carbonite builds up a buffer of extra copies. By the end of the simulations, the rate at which Carbonite produces new replicas approaches that of the oracle system.

5.3 How many replicas?

To formalize our intuition about the effect of extra replicas on maintenance cost and to understand how many extra replicas are necessary to avoid triggering repair following a transient failure, consider a simple Bernoulli process measuring R, the number of replicas available at a given moment, when there are $r > r_L$ total replicas. The availability of each node is a. Since repair is triggered when the number of available replicas is less than r_L , the probability that a new replica needs to be created is the probability

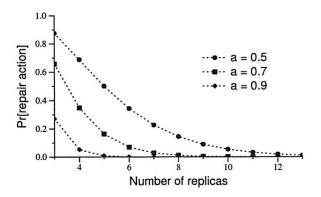


Figure 8: Additional redundancy must be created when the amount of live redundancy drops below the desired amount (3 replicas in this example). The probability of this happening depends solely on the average node availability a and the amount of durable redundancy. This graph shows the probability of a repair action as a function of the amount of durable redundancy, with a=0.5, a=0.7 and a=0.9 for a replication system.

that less than r_L replicas are available:

$$\Pr[R < r_L \mid r \text{ extant copies}] = \sum_{i=0}^{r_L-1} {r \choose i} a^i (1-a)^{r-i}.$$

This probability falls rapidly as r increases but it will never reach zero; there is always a chance that a replica must be created due to a large number of concurrent failures, regardless of how many replicas exist already. However, when a large number of replicas exists, it is extremely unlikely that enough replicas fail such that fewer than r_L are available.

By computing the Chernoff bound, it is possible to show that after the system has created $2r_L/a$ replicas, the probability of a new object creation is exponentially small. $2r_L/a$ is a rough (and somewhat arbitrary) estimate of when the probability of a new object creation is small enough to ignore. Figure 8 shows (on the y-axis) the probability that a new object must be created when an increasing number of replicas already exist. As r increases, the probability that a new replica needs to be created falls, and the algorithm creates replicas less frequently. As r approaches $2r_L/a$, the algorithm essentially stops creating replicas, despite not knowing the value of a.

This benefit is obtained only if returning replicas are reintegrated into the appropriate replica set, allowing more than r_L to be available with high probability. As a result, the cost of responding to transient failures will be nearly zero. Still, this system is more expensive than an oracle system that can distinguish between disk and transient failures. While the latter could maintain exactly r_L replicas, the former has to maintain approximately $2r_L/a$. The factor of 2/a difference in the cost is the penalty for

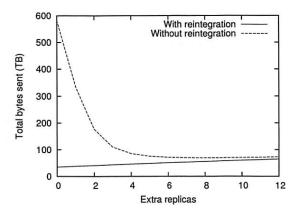


Figure 9: Total repair cost with extra replicas, and with and without reintegration after repair. Without reintegration, extra replicas reduce the rate at which repair is triggered and thus reduce maintenance cost; there is an optimal setting (here e=8). With reintegration, the cost is lowest if no extra replicas are used.

not distinguishing disk and transient failures.

5.4 Create replicas as needed

Given that the system tends towards creating $2r_L/a$ replicas in order to keep r_L of them available, it is tempting to create the entire set—not just r_L of them—when the object is first inserted into the system (Total Recall [3] uses a similar technique). However, this approach requires an accurate estimate for a to deliver good performance. If a is overestimated, the system quickly finds itself with less than r_L replicas after a string of transient failures and is forced to create additional copies. If a is underestimated, the system creates unneeded copies and wastes valuable resources. Carbonite is simplified by the fact that it does not need to measure or estimate a to create the "correct" number of replicas.

Another idea is to create not only enough copies to bring the number of available replicas back up to r_L , but also e additional copies beyond r_L (this is similar to Total Recall's lazy repair technique). Creating a batch of copies makes repair actions less frequent, but at the same time, causes more maintenance traffic than Carbonite. The work required to create additional replicas will be wasted if those replicas are lost due to disk failures before they are actually required. Carbonite, on the other hand, only creates replicas that are necessary to keep r_L replicas available. In other words, either Carbonite would eventually create the same number of replicas as a scheme that creates replicas in batches, or some replicas created in the batch were unnecessary: batch schemes do, at best, the same amount of work as Carbonite.

Figure 9 shows the bytes sent in a simulation experiment using a five-year synthetic trace with a = 0.88,

 $r_L=3$, and an average node lifetime of one year. The graph shows results for different values of e (in Total Recall, $e=r_H-r_L$) and for two different scenarios. In the scenario with reintegration, the system reintegrates all replicas as they return from transient failures. This scenario represents the behavior of Carbonite when e=0 and causes the least traffic.

In the scenario without reintegration, replicas that are unavailable when repair is triggered are not reintegrated into the replica set even if they do return. Total Recall behaves this way. Extra replicas give the system a short-term memory. Additional replicas increase the time until repair must be made (at which time failed replicas will be forgotten); during this time failed replicas can be reintegrated. Larger values of e give the system a longer memory but also put more data at risk of failure: on this synthetic trace, a value of e=8 is optimal. Taking advantage of returning replicas is simpler and more efficient than creating additional replicas: a system that reintegrates returning replicas will always make fewer copies than a system that does not and must replace forgotten replicas.

For systems that use erasure codes, there is an additional read cost since a complete copy of the object is needed in order to generate a new fragment [32]. The cost of reading a sufficient number of fragments prior to recreating a lost fragment can overwhelm the savings that erasure codes provide. A common approach is to amortize this cost by batching fragment creation but simply caching the object at the node responsible for repair is much more effective. A simulation contrasting both caching and batching (but both with reintegration) shows results similar to Figure 9: caching the object with a 7/14 erasure code uses 85% of the bandwidth that the optimal batching strategy would use.

5.5 Timeouts

A common approach to reduce transient costs is to use long timeouts, as suggested by Blake [4]. Timeouts are a heuristic to avoid misclassifying temporary failures as permanent: failures are considered to be permanent only when the corresponding node has not responded for some number of seconds. Longer timeouts reduce the number of misclassified transient failures and thus the number of repairs. On the other hand, a longer timeout also increases the latency between failure and repair in the event of a true disk failure; if additional permanent failures occur during this larger "window of vulnerability," data may be lost.

The goal of both reintegrating replicas and use of timeouts is to reduce the number of repairs without decreasing durability. Figure 7 demonstrates that reintegration is effective for Carbonite. However, it also illustrates that timeouts are important in systems without reintegration: on the PlanetLab trace, the timeout used is able to mask 87.7% of transient failures whereas it only masks 58.3% of transient failures on the Poisson trace. If replicas are reintegrated, what extra benefit does a timeout provide?

Timeouts are most effective when a significant percentage of the transient failures can be ignored, which is dependent on the downtime distribution. However, for durability to remain high, the expected node lifetime needs to be significantly greater than the timeout.

To evaluate this scenario where timeouts should have impact, we performed an experiment using a synthetic trace where we varied the repair threshold and the node timeout. Since the system would recognize nodes returning after a permanent failure and immediately expire all pending timeouts for these nodes, we assigned new identities to such nodes to allow long timeouts to expire normally.

Figure 10 shows the results of this simulation: (a) shows the total bytes sent as a function of timeout while (b) shows the durability at the end of the trace. As the length of the timeout increases past the average downtime, we observe a reduction in the number of bytes sent without a decrease in durability. However, as the timeout grows longer, durability begins to fall: the long timeout delays the point at which the system can begin repair, reducing the effective repair rate. Thus setting a timeout can reduce response to transient failures but its success depends greatly on its relationship to the downtime distribution and can in some instances reduce durability as well.

6 Implementing Carbonite

While the discussion of durability and efficient maintenance may be broadly applicable, in this section, we focus on our experience in implementing Carbonite in the context of distributed hash tables (DHTs).

In a DHT, each node is algorithmically assigned a portion of the total identifier space that it is responsible for maintaining. Carbonite requires that each node know the number of available replicas of each object for which it is responsible. The goal of monitoring is to allow the nodes to track the number of available replicas and to learn of objects that the node should be tracking but is not aware of. When a node n fails the new node n' that assumes responsibility of n's blocks begins tracking replica availability; monitored information is soft state and thus can be failed over to a "successor" relatively transparently.

Monitoring can be expensive: a node might have to contact every node in the scope of each object it holds. While developing two prototype implementations of Carbonite in the PlanetLab environment, we found it necessary to develop different techniques for monitoring: the monitoring problem is slightly different in systems that use distributed directories and those that use consistent hashing. Figure 11 illustrates the structures of these systems.

The Chord/DHash system [8, 9] served as the basis for our consistent hashing implementation. It uses a small

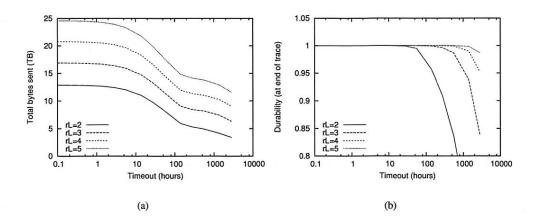


Figure 10: The impact of timeouts on bandwidth and durability on a synthetic trace. Figure 10(a) shows the number of copies created for various timeout values; (b) shows the corresponding object durability. In this trace, the expected downtime is about 29 hours. Longer timeouts allow the system to mask more transient failures and thus reduce maintenance cost; however, they also reduce durability.

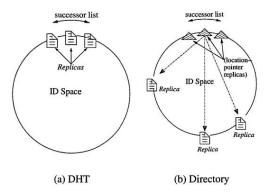


Figure 11: DHT- and Directory- Based Storage System Architectures.

scope and thus monitors a small number of nodes. DHash does not need to record the location of failed replicas: a node will return to the same place in the ring and thus the same replica sets, as long as it returns with the same logical identifier.

We used Oceanstore [20,30] and the BambooDHT [31] to develop a distributed directory system using large scope and random placement. Oceanstore must maintain pointers to all nodes that have ever held data for a given object and has a scope of N. Responsibility for keys is still assigned using consistent hashing: the pointer database for each key is replicated on the successors of the key. In this case, the location of objects is hard state. Unfortunately, it could be easy for this system to have very high monitoring costs: if each node communicate with every other node periodically, the resulting N^2 probe traffic may limit the system's scalability.

6.1 Monitoring consistent hashing systems

In systems that use a small scope, it is possible to make an end-to-end check that data is stored on the disk of each node. The naive way to do this is to arrange for nodes to repeatedly exchange key lists, but such an exchange would be extremely costly.

DHash uses a synchronization protocol based on Merkle trees [6] that takes advantage of the fact that most objects are typically correctly placed. In this common case, adjacent nodes store largely similar keys and two nodes can exchange a single message (containing a digest of the stored keys) to verify that they are synchronized.

Carbonite allows replicas to be placed anywhere in the placement scope. This flexibility lets the system avoid moving and replicating objects during most joins (until the system grows dramatically). However, it also causes the Merkle synchronization protocol to operate outside of its common case: adjacent nodes are no longer likely to store nearly identical sets of objects. In this environment the synchronizer "discovers" that nodes in the scope are missing objects each time it is run. Repeatedly exchanging this information can be costly: if the synchronization protocol runs once a minute, the cost of repeatedly transferring the 20-byte key of an 8 KB data object will exceed the cost of transferring the object itself to a newly joined node in about 8 hours.

To avoid this problem, each node maintains, for each object, a list of nodes in the scope without a copy of the object. The node uses this information to adjust its Merkle tree to avoid re-learning the information again during the next run of the synchronizer. For instance, when a node n synchronizes with a replica node n' that is known to be missing an object with key k, n leaves k out of the Merkle

tree used for synchronization: this prevents n' from reporting what n already knew. The amount of extra state needed to perform this optimization per object is small relative to the size of storing the object itself, and can be maintained lazily, unlike the object itself which is hard state.

6.2 Monitoring host availability

In a directory-style system, the same synchronization techniques just described can be used to monitor the directory itself (which is replicated on successor nodes); however, it is likely infeasible to explicitly monitor the liveness of objects themselves using the algorithm described above since two nodes are not likely to store the same keys. Instead, node availability can be monitored as a proxy for object availability. Node availability can be monitored using a multicast mechanism that propagates the liveness state of each node to each other node.

The DHT's routing tables are used to establish a unique spanning tree rooted at each node a $O(\log N)$ out-degree per node. Each node periodically broadcasts a heartbeat message to its children in the tree; this message includes a generation identifier that is randomly generated when the node is installed or reinstalled following a disk failure. The children rebroadcast the heartbeat to their children until it is received by all nodes.

Over time, each node expects to receive regular notification of node liveness. If a heartbeat is missed, the monitoring node triggers repair for every object stored on the newly down node. When a node returns and its generation identifier has not changed, the monitoring node can conclude that objects stored on that node are again accessible.

7 Related work

7.1 Replication analysis

The use of a birth-death data-loss model is a departure from previous analyses of reliability. Most DHT evaluations consider whether data would survive a single event involving the failure of many nodes [8,40]. This approach does not separate durability from availability, and does not consider the continuous bandwidth consumed by replacing replicas lost to disk failure.

The model and discussion in this paper is similar to contemporary work that looks at churn [37] and analyzes the expected object lifetime [29]. The birth-death model is a generalization of the calculations that predict the MTBF for RAID storage systems [26]. Owing to its scale, a distributed system has more flexibility to choose parameters such as the replication level and number of replica sets when compared to RAID systems.

Blake and Rodrigues argue that wide-area storage systems built on unreliable nodes cannot store a large amount of data [4]. Their analysis is based on the amount of data that a host can copy during its lifetime and mirrors our discussion of feasibility. We come to a different conclusion because we consider a relatively stable system membership where data loss is driven by disk failure, while they assumed a system with continual membership turnover.

The selection of a target replication level for surviving bursts differs from many traditional fault tolerant storage systems. Such systems, designed for single-site clusters, typically aim to continue operating despite some fixed number of failures and choose number of replicas so that a voting algorithm can ensure correct updates in the presence of partitions or Byzantine failures [5, 17, 23, 24, 33].

FAB [33] and Chain Replication [38] both consider how the number of possible replicas sets affects data durability. The two come to opposite conclusions: FAB recommends a small number of replica sets since more replica sets provide more ways for data to fail; chain replication recommends many replica sets to increase repair parallelism and thus reduce repair time. These observations are both correct: choosing a replica placement strategy requires balancing the probability of losing some data item during a simultaneous failure (by limiting the number of replica sets) and improving the ability of the system to tolerate a higher average failure rate (by increasing the number of replica sets and reconstruction parallelism).

Weatherspoon *et al* [39] studied the increased costs due to transient failures. Their results quantify the benefits of maintaining extra replicas in reducing these transient costs. However, their analysis focuses on systems that forget about extant replicas that exist when repair is initiated and do not discuss the benefits of reintegrating them.

7.2 Replicated systems

Replication has been widely used to reduce the risk of data loss and increase data availability in storage systems (e.g., RAID [26], System R duplex disks [16], Harp [23], xFS [2], Petal [21], DDS [17], GFS [15]). The algorithms traditionally used to create and maintain data redundancy are tailored for the environment in which these systems operate: well-connected hosts that rarely lose data or become unavailable. As a result they can maintain a small, fixed number of replicas and create a new replica immediately following a failure. This paper focuses on wide-area systems that are bandwidth-limited, where transient network failures are common, and where it is difficult to tell the difference between transient failures and disk failures.

Distributed databases [10], online disaster recovery systems such as Myriad [22], and storage systems [12, 13, 27] use replication and mirroring to distribute load and increase durability. These systems store mutable data and focus on the cost of propagating updates, a consideration not applicable to the immutable data we assume. In some cases, data is replicated between a primary and backup sites and further replicated locally at each site using RAID. Wide area recovery is initiated only after site

failure; individual disk failure can be repaired locally.

Total Recall is the system most similar to our work [3]. We borrow from Total Recall the idea that creating and tracking additional replicas can reduce the cost of transient failures. Total Recall's lazy replication keeps a fixed number of replicas and fails to reincorporate replicas that return after a transient failure if a repair had been performed. Total Recall also requires introspection or guessing to determine an appropriate high water mark that Carbonite can arrive at naturally.

Glacier [18] is a distributed storage system that uses massive replication to provide data durability across large-scale correlated failure events. The resulting tradeoffs are quite different from those of Carbonite, which is designed to handle a continuous stream of at small-scale failure events. For example, due to its high replication level, Glacier can afford very long timeouts and thus mask almost all transient failures.

8 Conclusions and future work

Inexpensive hardware and the increasing capacity of wide-area network links have spurred the development of applications that store a large amount of data on wide-area nodes. However, the feasibility of applications based on distributed storage systems is currently limited by the expense of maintaining data. This paper has described a set of techniques that allow wide-area systems to efficiently store and maintain large amounts of data.

These techniques have allowed us to develop and deploy prototypes of UsenetDHT [34], OverCite [35], and Antiquity [11]. These systems must store large amounts of data durably and were infeasible without the techniques we have presented. In the future, we hope to report on our long-term experience with these systems.

Acknowledgments The authors would like to thank Vivek Pai and Aaron Klingaman for their assistance in compiling the data used for the PlanetLab traces. This paper has benefited considerably from the comments of the anonymous reviewers and our shepherd, Larry Peterson.

References

- ANDERSEN, D. Improving End-to-End Availability Using Overlay Networks. PhD thesis, Massachusetts Institute of Technology, 2004
- [2] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *Proc. of the 15th ACM Symposium on Operating* System Principles (Dec. 1995).
- [3] BHAGWAN, R., TATI, K., CHENG, Y.-C., SAVAGE, S., AND VOELKER, G. M. Total Recall: System support for automated availability management. In Proc. of the 1st Symposium on Networked Systems Design and Implementation (Mar. 2004).
- [4] BLAKE, C., AND RODRIGUES, R. High availability, scalable storage, dynamic peer networks: Pick two. In Proc. of the 9th Workshop on Hot Topics in Operating Systems (May 2003), pp. 1–6.

- [5] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20, 4 (2002), 398–461.
- [6] CATES, J. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.
- [7] DABEK, F. A Distributed Hash Table. PhD thesis, Massachusetts Institute of Technology, 2005.
- [8] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In Proc. of the 18th ACM Symposium on Operating System Principles (Oct. 2001)
- [9] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In Proc. of the 1st Symposium on Networked Systems Design and Implementation (Mar. 2004).
- [10] DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In Proc. of the 6th ACM Symposium on Principles of Distributed Computing (1987), pp. 1-12.
- [11] EATON, P., WEATHERSPOON, H., AND KUBIATOWICZ, J. Efficiently binding data to owners in distributed content-addressable storage systems. In *Proc. of the 3rd International Security in Stor*age Workshop (Dec. 2005).
- [12] EMC. Centera—content addressed storage system. http:// www.emc.com/products/systems/centera.jsp. Last accessed March 2006.
- [13] EMC. Symmetrix remote data facility. http://www.emc. com/products/networking/srdf.jsp. Last accessed March 2006.
- [14] FREEDMAN, M. J., LAKSHMINARAYANAN, K., RHEA, S., AND STOICA, I. Non-transitive connectivity and DHTs. In Proc. of the 2nd Workshop on Real Large Distributed Systems (Dec. 2005).
- [15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proc. of the 2003 19th ACM Symposium on Operating System Principles (Oct. 2003).
- [16] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. ACM Computing Surveys 13, 2 (1981), 223–242.
- [17] GRIBBLE, S., BREWER, E., HELLERSTEIN, J., AND CULLER, D. Scalable, distributed data structures for internet service construction. In Proc. of the 4th Symposium on Operating Systems Design and Implementation (Oct. 2004).
- [18] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier. Highly durable, decentralized storage despite massive correlated failures. In Proc. of the 2nd Symposium on Networked Systems Design and Implementation (May 2005).
- [19] KLEINROCK, L. Queueing Systems, Volume I: Theory. John Wiley & Sons, Jan. 1975.
- [20] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATH-ERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In Proc. of the 9th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (Nov. 2000), pp. 190–201.
- [21] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In Proc. of the 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (1996), pp. 84–92.

- [22] LEUNG, S.-T. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. Myriad: Cost-effective disaster tolerance. In Proc. of the 1st USENIX Conference on File and Storage Technologies (Jan. 2002).
- [23] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp file system. In Proc. of the 13th ACM Symposium on Operating System Principles (Oct. 1991), pp. 226–38.
- [24] LITWIN, W., AND SCHWARZ, T. LH* RS: A high-availability scalable distributed data structure using reed solomon codes. In Proc. of the 2000 ACM SIGMOD Intl. Conference on Management of Data (May 2000), pp. 237–248.
- [25] PARK, K. S., AND PAI, V. CoMon: a mostly-scalable monitoring system for PlanetLab. ACM SIGOPS Operating Systems Review 40, 1 (Jan. 2006), 65–74. http://comon.cs.princeton. edu/.
- [26] PATTERSON, D., GIBSON, G., AND KATZ, R. A case for redundant arrays of inexpensive disks (RAID). In Proc. of the ACM SIGMOD International Conference on Management of Data (June 1988).
- [27] PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. Snapmirror: File system based asynchronous mirroring for disaster recovery. In *Proc. of the 1st* USENIX Conference on File and Storage Technologies (Jan. 2002).
- [28] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A blueprint for introducing disruptive technology into the Internet. In Proc. of the First ACM Workshop on Hot Topics in Networks (Oct. 2002). http://www.planet-lab.org.
- [29] RAMABHADRAN, S., AND PASQUALE, J. Analysis of longrunning replicated systems. In Proc. of the 25th IEEE Annual Conference on Computer Communications (INFOCOM) (Apr. 2006).
- [30] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In Proc. of the 2nd USENIX Conference on File and Storage Technologies (Apr. 2003).
- [31] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In Proc. of the 2004 Usenix Annual Technical Conference (June 2004).
- [32] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In Proc. of the 4th International Workshop on Peer-to-Peer Systems (Feb. 2005).
- [33] SAITO, Y., FRŒLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: building distributed enterprise disk arrays from commodity components. In Proc. of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, 2004), ACM Press, pp. 48–58.
- [34] SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead Usenet server. In Proc. of the 3rd International Workshop on Peer-to-Peer Systems (Feb. 2004).
- [35] STRIBLING, J., COUNCILL, I. G., LI, J., KAASHOEK, M. F., KARGER, D. R., MORRIS, R., AND SHENKER, S. OverCite: A cooperative digital research library. In Proc. of the 4th International Workshop on Peer-to-Peer Systems (Feb. 2005).
- [36] STRIBLING, J., LI, J., COUNCILL, I. G., KAASHOEK, M. F., AND MORRIS, R. Exploring the design of multi-site web services using the OverCite digital library. In Proc. of the 3rd Symposium on Networked Systems Design and Implementation (May 2006).
- [37] TATI, K., AND VOELKER, G. M. On object maintenance in peer-to-peer systems. In Proc. of the 5th International Workshop on Peer-to-Peer Systems (Feb. 2006).
- [38] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In Proc. of the 6th Symposium on Operating Systems Design and Implementation (Dec. 2004).

- [39] WEATHERSPOON, H., CHUN, B.-G., SO, C. W., AND KUBIA-TOWICZ, J. Long-term data maintenance in wide-area storage systems: A quantitative approach. Tech. Rep. UCB//CSD-05-1404, U. C. Berkeley, July 2005.
- [40] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).

PRACTI Replication

Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, Jiandan Zheng University of Texas at Austin*

Abstract

We present PRACTI, a new approach for large-scale replication. PRACTI systems can replicate or cache any subset of data on any node (Partial Replication), provide a broad range of consistency guarantees (Arbitrary Consistency), and permit any node to send information to any other node (Topology Independence). A PRACTI architecture yields two significant advantages. First, by providing all three PRACTI properties, it enables better trade-offs than existing mechanisms that support at most two of the three desirable properties. The PRACTI approach thus exposes new points in the design space for replication systems. Second, the flexibility of PRACTI protocols simplifies the design of replication systems by allowing a single architecture to subsume a broad range of existing systems and to reduce development costs for new ones. To illustrate both advantages, we use our PRACTI prototype to emulate existing server replication, client-server, and object replication systems and to implement novel policies that improve performance for mobile users, web edge servers, and grid computing by as much as an order of magnitude.

1 Introduction

This paper describes PRACTI, a new data replication approach and architecture that can reduce replication costs by an order of magnitude for a range of large-scale systems and also simplify the design, development, and deployment of new systems.

Data replication is a building block for many large-scale distributed systems such as mobile file systems, web service replication systems, enterprise file systems, and grid replication systems. Because there is a fundamental trade-off between performance and consistency [22] as well as between availability and consistency [9, 31], systems make different compromises among these factors by implementing different placement policies, consistency policies, and topology policies for different environments. Informally, placement policies such as demand-caching, prefetching, or replicate-all define which nodes store local copies of

which data, *consistency policies* such as sequential [21] or causal [16] define which reads must see which writes, and *topology policies* such as client-server, hierarchy, or ad-hoc define the paths along which updates flow.

This paper argues that an ideal replication framework should provide all three PRACTI properties:

- Partial Replication (PR) means that a system can place any subset of data and metadata on any node. In contrast, some systems require a node to maintain copies of all objects in all volumes they export [26, 37, 39].
- Arbitrary Consistency (AC) means that a system can
 provide both strong and weak consistency guarantees
 and that only applications that require strong guarantees pay for them. In contrast, some systems can only
 enforce relatively weak coherence guarantees and can
 make no guarantees about stronger consistency properties [11, 29].
- Topology Independence (TI) means that any node can exchange updates with any other node. In contrast, many systems restrict communication to clientserver [15, 18, 25] or hierarchical [4] patterns.

Although many existing systems can each provide two of these properties, we are aware of no system that provides all three. As a result, systems give up the ability to exploit locality, support a broad range of applications, or dynamically adapt to network topology.

This paper presents the first replication architecture to provide all three PRACTI properties. The protocol draws on key ideas of existing protocols but recasts them to remove the deeply-embedded assumptions that prevent one or more of the properties. In particular, our design begins with log exchange mechanisms that support a range of consistency guarantees and topology independence but that fundamentally assume full replication [26, 37, 39]. To support partial replication, we extend the mechanisms in two simple but fundamental ways.

1. In order to allow partial replication of data, our design separates the control path from the data path by separating invalidation messages that identify what has changed from body messages that encode the changes to the contents of files. Distinct invalidation messages are widely used in hierarchical caching systems, but we demonstrate how to use them in topology-independent systems: we develop explicit synchronization rules to enforce consistency despite multiple streams of information, and we introduce general

^{*}This work was supported in part by the National Science Foundation (CNS-0411026, SCI-0438314), the Texas Advanced Technology Program, and an IBM University Partnership Award. Gao's present affiliation: Oracle, Inc. Venkataramani's present affiliation: University of Massachusettes, Amherst. Yalagandula's present affiliation: Hewlett Packard Labs.

mechanisms for handling demand read misses.

2. In order to allow partial replication of update metadata, we introduce *imprecise invalidations*, which allow a single invalidation to summarize a set of invalidations. Imprecise invalidations provide cross-object consistency in a scalable manner: each node incurs storage and bandwidth costs proportional to the size of the data sets in which it is interested. For example, a node that is interested in one set of objects A but not another set B, can receive precise invalidations for objects in A along with an imprecise invalidation that summarizes omitted invalidations to objects in B. The imprecise invalidation then serves as a placeholder for the omitted updates both in the node's local storage and in the logs of updates the node propagates to other nodes.

We construct and evaluate a prototype using a range of policies and workloads. Our primary conclusion is that by simultaneously supporting the three PRACTI properties, PRACTI replication enables better trade-offs for system designers than possible with existing mechanisms. For example, for some workloads in our mobile storage and grid computing case studies, our system dominates existing approaches by providing more than an order of magnitude better bandwidth and storage efficiency than full replication AC-TI replicated server systems, by providing more than an order of magnitude better synchronization delay compared to topology constrained PR-AC hierarchical systems, and by providing consistency guarantees not achievable by limited consistency PR-TI object replication systems.

More broadly, we argue that PRACTI protocols can simplify the design of replication systems. At present, because mechanisms and policies are entangled, when a replication system is built for a new environment, it must often be built from scratch or must modify existing mechanisms to accommodate new policy trade-offs. In contrast, our system can be viewed as a replication microkernel that defines a common substrate of core mechanisms over which a broad range of systems can be constructed by selecting appropriate policies. For example, in this study we use our prototype both to emulate existing server replication, client-server, and object replication systems and to implement novel policies to support mobile users, web edge servers, and grid computing.

In summary, this paper makes four contributions. First, it defines the PRACTI paradigm and provides a taxonomy for replication systems that explains why existing replication architectures fall short of ideal. Second, it describes the first replication protocol to simultaneously provide all three PRACTI properties. Third, it provides a prototype PRACTI replication toolkit that cleanly separates mechanism from policy and thereby allows nearly arbitrary replication, consistency, and topology policies.

Fourth, it demonstrates that PRACTI replication offers decisive practical advantages compared to existing approaches.

Section 2 revisits the design of existing systems in light of the PRACTI taxonomy. Section 3 describes our protocol for providing PRACTI replication, and Section 4 experimentally evaluates the prototype. Finally, Section 5 surveys related work, and Section 6 highlights our conclusions.

2 Taxonomy and challenges

In order to put the PRACTI approach in perspective, this section examines existing replication architectures and considers why years of research exploring many different replication protocols have failed to realize the goal of PRACTI replication.

Note that the requirements for supporting flexible consistency guarantees are subtle, and Section 3.3 discusses the full range of flexibility our protocol provides. PRACTI replication should support both the weak coherence-only guarantees acceptable to some applications and the stronger consistency guarantees required by others. Note that consistency semantics constrain the order that updates across multiple objects become observable to nodes in the system while coherence semantics are less restrictive in that they only constrain the order that updates to a single object become observable but do not additionally constrain the ordering of updates across multiple locations. (Hennessy and Patterson discusses the distinction between consistency and coherence in more detail [12].) For example, if a node n1 updates object A and then object B and another node n2 reads the new version of B, most consistency semantics would ensure that any subsequent reads by n2 see the new version of A, while most coherence semantics would permit a read of A to return either the new or old version.

PRACTI Taxonomy. The PRACTI paradigm defines a taxonomy for understanding the design space for replication systems as illustrated in Figure 1. As the figure indicates, many existing replication systems can be viewed as belonging to one of four protocol families, each of which provides at most two of the PRACTI properties.

Server replication systems like Replicated Dictionary [37] and Bayou [26] provide log-based peer-to-peer update exchange that allows any node to send updates to any other node (TI) and that consistently orders writes across all objects. Lazy Replication [19] and TACT [39] use this approach to provide a wide range of tunable consistency guarantees (AC). Unfortunately, these protocols fundamentally assume full replication: all nodes store all data from any volume they export and all nodes receive all updates. As a result, these systems are unable to exploit workload locality to efficiently use networks

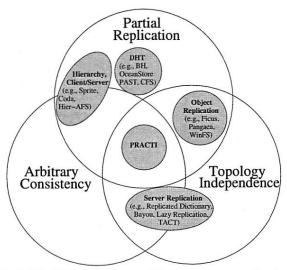


Fig. 1: The PRACTI taxonomy defines a design space for classifying families of replication systems.

and storage, and they may be unsuitable for devices with limited resources.

Client-server systems like Sprite [25] and Coda [18] and hierarchical caching systems like hierarchical AFS [24] permit nodes to cache arbitrary subsets of data (PR). Although specific systems generally enforce a set consistency policy, a broad range of consistency guarantees are provided by variations of the basic architecture (AC). However, these protocols fundamentally require communication to flow between a child and its parent. Even when systems permit limited client-client communication for cooperative caching, they must still serialize control messages at a central server for consistency [5]. These restricted communication patterns (1) hurt performance when network topologies do not match the fixed communication topology or when network costs change over time (e.g., in environments with mobile nodes), (2) hurt availability when a network path or node failure disrupts a fixed communication topology, and (3) limit sharing during disconnected operation when a set of nodes can communicate with one another but not with the rest of the system.

DHT-based storage systems such as BH [35], PAST [28], and CFS [6] implement a specific—if sophisticated—topology and replication policy: they can be viewed as generalizations of client-server systems where the server is split across a large number of nodes on a per-object or per-block basis for scalability and replicated to multiple nodes for availability and reliability. This division and replication, however, introduce new challenges for providing consistency. For example, the Pond OceanStore prototype assigns each object to a set of primary replicas that receive all updates for the object, uses an agreement protocol to coordinate these servers for per-object coherence, and does not attempt to

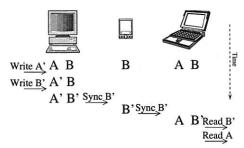


Fig. 2: Naive addition of PR to an AC-TI log exchange protocol fails to provide consistency.

provide cross-object consistency guarantees [27].

Object replication systems such as Ficus [11], Pangaea [29], and WinFS [23] allow nodes to choose arbitrary subsets of data to store (PR) and arbitrary peers with whom to communicate (TI). But, these protocols enforce no ordering constraints on updates across multiple objects, so they can provide coherence but not consistency guarantees. Unfortunately, reasoning about the corner cases of consistency protocols is complex, so systems that provide only weak consistency or coherence guarantees can complicate constructing, debugging, and using the applications built over them. Furthermore, support for only weak consistency may prevent deployment of applications with more stringent requirements.

Why is PRACTI hard? It is surprising that despite the disadvantages of omitting any of the PRACTI properties, no system provides all three. Our analysis suggests that these limitations are fundamental to these existing protocol families: the assumption of full replication is deeply embedded in the core of server replication protocols; the assumption of hierarchical communication is fundamental to client-server consistency protocols; careful assignment of key ranges to nodes is central to the properties of DHTs; and the lack of consistency is a key factor in the flexibility of object replication systems.

To understand why it is difficult for existing architectures to provide all three PRACTI properties, consider Figure 2's illustration of a naive attempt to add PR to a AC-TI server replication protocol like Bayou. Suppose a user's desktop node stores all of the user's files, including files A and B, but the user's palmtop only stores a small subset that includes B but not A. Then, the desktop issues a series of writes, including a write to file A (making it A') followed by a write to file B (making it B'). When the desktop and palmtop synchronize, for PR, the desktop sends the write of B but not the write of A. At this point, everything is OK: the palmtop and desktop have exactly the data they want, and reads of local data provide a consistent view of the order that writes occurred. But for TI, we not only have to worry about local reads but also propagation of data to other nodes. For instance, suppose that the user's laptop, which also stores all of the user's files including both A and B, synchronizes with

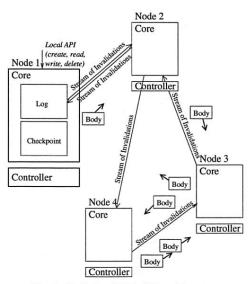


Fig. 3: High level PRACTI architecture.

the palmtop: the palmtop can send the write of B but not the write of A. Unfortunately, the laptop now can present an inconsistent view of data to a user or application. In particular, a sequence of reads at the laptop can return the new version of B and then return the old version of A, which is inconsistent with the writes that occurred at the desktop under causal [16] or even the weaker FIFO consistency [22].

This example illustrates the broader, fundamental challenge: supporting flexible consistency (AC) requires careful ordering of how updates propagate through the system, but consistent ordering becomes more difficult if nodes communicate in ad-hoc patterns (TI) or if some nodes know about updates to some objects but not other objects (PR).

Existing systems resolve this dilemma in one of three ways. The full replication of AC-TI replicated server systems ensures that all nodes have enough information to order all updates. Restricted communication in PR-AC client-server and hierarchical systems ensures that the root server of a subtree can track what information is cached by descendents; the server can then determine which invalidations it needs to propagate down and which it can safely omit. Finally, PR-TI object replication systems simply give up ability to consistently order writes to different objects and instead allow inconsistencies such as the one just illustrated.

3 PRACTI replication

Figure 3 shows the high-level architecture of our implementation of a PRACTI protocol.

Node 1 in the figure illustrates the main local data structures of each node. A node's Core embodies the protocol's mechanisms by maintaining a node's local state. Applications access data stored in the local core via the per-node Local API for creating, reading, writing, and

deleting objects. These functions operate the local node's Log and Checkpoint: modifications are appended to the log and then update the checkpoint, and reads access the random-access checkpoint. To support partial replication policies, the mechanisms allow each node to select an arbitrary subset of the system's objects to store locally, and nodes are free to change this subset at any time (e.g., to implement caching, prefetching, hoarding, or replicate-all). This local state allows a node to satisfy requests to read valid locally-stored objects without needing to communicate with other nodes.

To handle read misses and to push updates between nodes, cores use two types of communication as illustrated in the figure—causally ordered *Streams of Invalidations* and unordered *Body* messages. The protocol for sending streams of invalidations is similar to Bayou's [26] log exchange protocol, and it ensures that each node's log and checkpoint always reflect a causally consistent view of the system's data. But it differs from existing log exchange protocols in two key ways:

- Separation of invalidations and bodies. Invalidation streams notify a receiver that writes have occurred, but separate body messages contain the contents of the writes. A core coordinates these separate sources of information to maintain local consistency invariants. This separation supports partial replication of data—a node only needs to receive and store bodies of objects that interest it.
- 2. Imprecise invalidations. Although the invalidation streams each logically contain a causally consistent record of all writes known to the sender but not the receiver, nodes can omit sending groups of invalidations by instead sending imprecise invalidations. Whereas traditional precise invalidations describe the target and logical time of a single write, an imprecise invalidation can concisely summarize a set of writes over an interval of time across a set of target objects. Thus, a single imprecise invalidation can replace a large number of precise invalidations and thereby support partial replication of metadata—a node only needs to receive traditional precise invalidations and store per-object metadata for objects that interest it.

Imprecise invalidations allow nodes to maintain consistency invariants despite partial replication of metadata and despite topology independence. In particular, they serve as placeholders in a receiver's log to ensure that there are no causal gaps in the log a node stores and transmits to other nodes. Similarly, just as a node tracks which objects are *INVALID* so it can block a read to an object that has been invalidated but for which the corresponding body message has not been received, a node tracks which sets of objects are *IMPRECISE* so it can block a read to an object that

has been targeted by an imprecise invalidation and for which the node therefore may not know about the most recent write.

The mechanisms just outlined, embodied in a node's *Core*, allow a node to store data for any subsets of objects, to store per-object metadata for any subset of objects, to receive precise invalidations for any subset of objects from any node, and to receive body messages for any subset of objects from any node. Given these mechanisms, a node's *Controller* embodies a system's replication and topology policies by directing communication among nodes. A node's controller (1) selects which nodes should send it invalidations and, for each invalidation stream subscription, specifies subsets of objects for which invalidations should be precise, (2) selects which nodes to prefetch bodies from and which bodies to prefetch, and (3) selects which node should service each demand read miss.

These mechanisms also support flexible consistency via a variation of the TACT [39] interface, which allows individual read and write requests to specify the semantics they require. By using this interface, applications that require weak guarantees can minimize performance [22] and availability [9] overheads while applications that require strong guarantees can get them.

The rest of this section describes the design in more detail. It first explains how our system's log exchange protocol separates invalidation and body messages. It then describes how imprecise invalidations allow the log exchange protocol to partially replicate invalidations. Next, it discusses the crosscutting issue of how to provide flexible consistency. After that, it describes several novel features of our prototype that enable it to support the broadest range of policies.

3.1 Separation of invalidations and bodies

As just described, nodes maintain their local state by exchanging two types of updates: ordered streams of invalidations and unordered body messages. *Invalidations* are metadata that describe writes; each contains an object ID¹ and logical time of a write. A write's logical time is assigned at the local interface that first receives the write, and it contains the current value of the node's Lamport clock [20] and the node's ID. Like invalidations, *body messages* contain the write's object ID and logical time, but they also contain the actual contents of the write.

The protocol for exchanging updates is simple.

As illustrated for node 1 in Figure 3, each node maintains a log of the invalidations it has received sorted by logical time. And, for random access, each node stores bodies in a checkpoint indexed by object ID.

- Invalidations from a log are sent via a causally-ordered stream that logically contains all invalidations known to the sender but not to the receiver. As in Bayou, nodes use version vectors to summarize the contents of their logs in order to efficiently identify which updates in a sender's log are needed by a receiver [26].
- A receiver of an invalidation inserts the invalidation into its sorted log and updates its checkpoint. Checkpoint update of the entry for object ID entails marking the entry INVALID and recording the logical time of the invalidation. Note that checkpoint update for an incoming invalidation is skipped if the checkpoint entry already stores a logical time that is at least as high as the logical time of the incoming invalidation.
- A node can send any body from its checkpoint to any other node at any time. When a node receives a body, it updates its checkpoint entry by first checking to see if the entry's logical time matches the body's logical time and, if so, storing the body in the entry and marking the entry VALID.

Rationale. Separating invalidations from bodies provides topology-independent protocol that supports both arbitrary consistency and partial replication.

Supporting arbitrary consistency requires a node to be able to consistently order all writes. Log-based invalidation exchange meets this need by ensuring three crucial properties [26]. First the prefix property ensures that a node's state always reflects a prefix of the sequence of invalidations by each node in the system, i.e., if a node's state reflects the ith invalidation by some node n in the system, then the node's state reflects all earlier invalidations by n. Second, each node's local state always reflects a causally consistent [16] view of all invalidations that have occurred. This property follows from the prefix property and from the use of Lamport clocks to ensure that once a node has observed the invalidation for write w, all of its subsequent local writes' logical timestamps will exceed w's. Third, the system ensures eventual consistency: all connected nodes eventually agree on the same total order of all invalidations. This combination of properties provides the basis for a broad range of tunable consistency semantics using standard techniques [39].

At the same time, this design supports partial replication by allowing bodies to be sent to or stored on any node at any time. It supports arbitrary body replication policies including demand caching, push-caching, prefetching, hoarding, pre-positioning bodies according to a global placement policy, or push-all.

Design issues. The basic protocol adapts well-understood log exchange mechanisms [26, 37]. But, the separation of invalidations and bodies raises two new issues: (1) coordinating disjoint streams of invalidations and bodies and (2) handling reads of invalid data.

¹For simplicity, we describe the protocol in terms of full-object writes. For efficiency, our implementation actually tracks checkpoint state, invalidations, and bodies on arbitrary byte ranges.

The first issue is how to coordinate the separate body messages and invalidation streams to ensure that the arrival of out-of-order bodies does not break the consistency invariants established by the carefully ordered invalidation log exchange protocol. The solution is simple: when a node receives a body message, it does not apply that message to its checkpoint until the corresponding invalidation has been applied. A node therefore buffers body messages that arrive "early." As a result, the checkpoint is always consistent with the log, and the flexible consistency properties of the log [39] extend naturally to the checkpoint despite its partial replication.

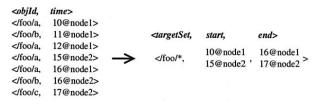
The second issue is how to handle demand reads at nodes that replicate only a subset of the system's data. The core mechanism supports a wide range of policies: by default, the system blocks a local read request until the requested object's status is *VALID*. Of course, to ensure liveness, when an *INVALID* object is read, an implementation should arrange for someone to send the body. Therefore, when a local read blocks, the core notifies the controller. The controller can then implement any policy for locating and retrieving the missing data such as sending the request up a static hierarchy (i.e., ask your parent or a central server), querying a separate centralized [8] or DHT-based [35] directory, using a hint-based search strategy, or relying on a push-all strategy [26, 37] (i.e., just wait and the data will come.)

3.2 Partial replication of invalidations

Although separation of invalidations from bodies supports partial replication of bodies, for true partial replication the system must not require all nodes to see all invalidations or to store metadata for each object. Exploiting locality is fundamental to replication in largescale systems, and requiring full replication of metadata would prevent deployment of a replication system for a wide range of environments, workloads, and devices. For example, consider palmtops caching data from an enterprise file system with 10,000 users and 10,000 files per user: if each palmtop were required to store 100 bytes of per-object metadata, then 10GB of storage would be consumed on each device. Similarly, if the palmtops were required to receive every invalidation during log exchange and if an average user issued just 100 updates per day, then invalidations would consume 100MB/day of bandwidth to each device.

To support true partial replication, invalidation streams *logically* contain all invalidations as described in Section 3.1, but in *reality* they omit some by replacing them with *imprecise invalidations*.

As Figure 4 illustrates, an imprecise invalidation is a conservative summary of several standard or *precise invalidations*. Each imprecise invalidation has a *targetSet* of objects, *start* logical time, and an *end* logical time, and



Precise Invalidations

Imprecise Invalidation

Fig. 4: Example imprecise invalidation.

it means "one or more objects in targetSet were updated between start and end." An imprecise invalidation must be conservative: each precise invalidation that it replaces must have its objId included in targetSet and must have its logical time included between start and end, but for efficient encoding targetSet may include additional objects. In our prototype, the targetSet is encoded as a list of subdirectories and the start and end times are partial version vectors with an entry for each node whose writes are summarized by the imprecise invalidation.

A node reduces its bandwidth requirements by subscribing to receive precise invalidations only for desired subsets of data and receiving imprecise invalidations for the rest. And a node saves storage by tracking per-object state only for desired subsets of data and tracking coarsegrained bookkeeping information for the rest.

Processing imprecise invalidations. When a node receives imprecise invalidation I, it inserts I into its log and updates its checkpoint. For the log, imprecise invalidations act as placeholders to ensure that the omitted precise invalidations do not introduce causal gaps in the log that a node stores locally or in the streams of invalidations that a node transmits to other nodes.

Tracking the effects of imprecise invalidations on a node's checkpoint must address four related problems:

- For consistency, a node must *logically* mark all objects targeted by a new imprecise invalidation as *INVALID*.
 This action ensures that if a node tries to read data that may have been updated by an omitted write, the node can detect that information is missing and block the read until the missing information has been received.
- For liveness, a node must be able to unblock reads for an object once the per-object state is brought up to date (e.g., when a node receives the precise invalidations that were summarized by an imprecise invalidation.)
- 3. For space efficiency, a node should not have to store per-object state for all objects. As the example at the start of this subsection illustrates, doing so would significantly restrict the range of replication policies, devices, and workloads that can be accommodated.
- 4. For processing efficiency, a node should not have to iterate across all objects encompassed by *targetSet* to apply an imprecise invalidation.

To meet these requirements, rather than track the effects of imprecise invalidations on individual objects, nodes keep bookkeeping information on groups of objects called *Interest Sets*. In particular, each node independently partitions the object ID space into one or more interest sets and decides whether to store per-object state on a per-interest set basis. A node tracks whether each interest set is *PRECISE* (per-object state reflects all invalidations) or *IMPRECISE* (per-object state is not stored or may not reflect all precise invalidations) by maintaining two pieces of state.

- Each node maintains a global variable currentVV, which is a version vector encompassing the highest timestamp of any invalidation (precise or imprecise) applied to any interest set.
- Each node maintains for each interest set IS the variable IS.lastPreciseVV, which is the latest version vector for which IS is known to be PRECISE.

If IS.lastPreciseVV = currentVV, then interest set IS has not missed any invalidations and it is PRECISE.

In this arrangement, applying an imprecise invalidation *I* to an interest set *IS* merely involves updating two variables—the global *currentVV* and the interest set's *IS*.*lastPreciseVV*. In particular, a node that receives imprecise invalidation *I* always advances *currentVV* to include *I's end* logical time because after applying *I*, the system's state may reflect events up to *I.end*. Conversely, the node only advances *IS*.*lastPreciseVV* to the latest time for which *IS* has missed no invalidations.

This per-interest set state meets the four requirements listed above.

- By default, a read request blocks until the interest set in which the object lies is PRECISE and the object is VALID. This blocking ensures that reads only observe the checkpoint state they would have observed if all invalidations were precise and therefore allows nodes to enforce the same consistency guarantees as protocols without imprecise invalidations.
- 2. For liveness, the system must eventually unblock waiting reads. The core signals the controller when a read of an *IMPRECISE* interest set blocks, and the controller is responsible for arranging for the missing precise invalidations to be sent. When the missing invalidations arrive, they advance *IS.lastPreciseVV*. The algorithm for processing invalidations guarantees that any interest set *IS* can be made *PRECISE* by receiving a sequence *S* of invalidations from *IS.lastPreciseVV* to currentVV if *S* is causally sorted and includes all precise invalidations targeting *IS* in that interval.
- Storage is limited: each node only needs to store perobject state for data currently of interest to that node. Thus, the total metadata state at a node is proportional to the number of objects of interest plus the number

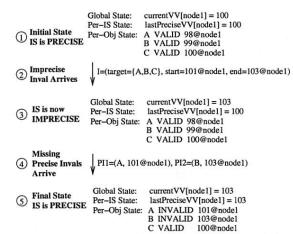


Fig. 5: Example of maintaining interest set state. For clarity, we only show node1's elements of *currentVV* and *lastPreciseVV*.

- of interest sets. Note that our implementation allows a node to dynamically repartition its data across interest sets as its locality patterns change.
- Imprecise invalidations are efficient to apply, requiring work that is proportional to the number of interest sets at the receiver rather than the number of summarized invalidations.

Example. The example in Figure 5 illustrates the maintenance of interest set state. Initially, (1) interest set IS is PRECISE and objects A, B, and C are VALID. Then, (2) an imprecise invalidation I arrives. I (3) advances currentVV but not IS.lastPreciseVV, making IS IMPRECISE. But then (4) precise invalidations PII and PI2 arrive on a single invalidation channel from another node. (5) These advance IS.lastPreciseVV, and in the final state IS is PRECISE, A and B are INVALID, and C is VALID.

Notice that although the node never receives a precise invalidation with time 102@node1, the fact that a single incoming stream contains invalidations with times 101@node1 and 103@node1 allows it to infer by the prefix property that no invalidation at time 102@node1 occurred, and therefore it is able to advance IS.lastPrecise-VV to make IS PRECISE.

3.3 Consistency: Approach and costs

Enforcing cache consistency entails fundamental tradeoffs. For example the CAP dilemma states that a replication system that provides sequential Consistency cannot simultaneously provide 100% Availability in an environment that can be Partitioned [9, 31]. Similarly, Lipton and Sandberg describe fundamental consistency v. performance trade-offs [22].

A system that seeks to support arbitrary consistency must therefore do two things. First, it must allow a range of consistency guarantees to be enforced. Second, it must ensure that workloads only pay for the consistency guarantees they actually need. Providing flexible guarantees. Discussing the semantic guarantees of large-scale replication systems requires careful distinctions along several dimensions. *Consistency* constrains the order that updates across multiple memory locations become observable to nodes in the system, while *coherence* constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across multiple locations [12]. *Staleness* constrains the realtime delay from when a write completes until it becomes observable. Finally, *conflict resolution* [18, 34] provides ways to cope with cases where concurrent reads and writes at different nodes conflict.

Our protocol provides considerable flexibility along all four of these dimensions.

With respect to consistency and staleness, it provides a range of traditional guarantees such as the relatively weak constraints of causal consistency [16, 20] or delta coherence [32], to the stronger constraints of sequential consistency [21] or linearizability [13]. Further, it provides a continuous range of guarantees between causal consistency, sequential consistency, and linearizability by supporting TACT's order error for bounding inconsistency and temporal error for bounding staleness [39]. Because our design uses a variation of peer-to-peer log exchange [26, 37], adapting flexible consistency techniques from the literature is straightforward.

With respect to coherence, although our default read interface enforces causal consistency, the interface allows programs that do not demand cross-object consistency to issue *imprecise reads*. Imprecise reads may achieve higher availability and performance than precise reads because they can return without waiting for an interest set to become *PRECISE*. Imprecise reads thus observe causal coherence (causally coherent ordering of reads and writes for any individual item) rather than causal consistency (causally consistent ordering of reads and writes across all items.)

With respect to conflict resolution, our prototype provides an interface for detecting and resolving write-write conflicts according to application-specific semantics [18, 26]. In particular, nodes log conflicting concurrent writes that they detect in a way that guarantees that all nodes that are *PRECISE* for an interest set will eventually observe the same sequence of conflicting writes for that interest set. The nodes then provide an interface for programs or humans to read these conflicting writes and to issue new compensating transactions to resolve the conflicts.

Costs of consistency. PRACTI protocols should ensure that workloads only pay for the semantic guarantees they need. Our protocol does so by distinguishing the availability and response time costs paid by read and

write requests from the bandwidth overhead paid by invalidation propagation.

The read interface allows each read to specify its consistency and staleness requirements. Therefore, a read does not block unless *that read* requires the local node to gather more recent invalidations and updates than it already has. Similarly, most writes complete locally, and a write only blocks to synchronize with other nodes if *that write* requires it. Therefore, as in TACT [39], the performance/availability versus consistency dilemmas are resolved on a per-read, per-write basis.

Conversely, all invalidations that propagate through the system carry sufficient information that a later read can determine what missing updates must be fetched to ensure the consistency or staleness level the read demands. Therefore, the system may pay an extra cost: if a deployment never needs strong consistency, then our protocol may propagate some bookkeeping information that is never used. We believe this cost is acceptable for two reasons: (1) other features of the design—separation of invalidations from bodies and imprecise invalidations—minimize the amount of extra data transferred; and (2) we believe the bandwidth costs of consistency are less important than the availability and response time costs. Experiments in Section 4 quantify these bandwidth costs, and we argue that they are not significant.

3.4 Additional features

Three novel aspects of our implementation further our goal of constructing a flexible framework that can accommodate the broadest range of policies. First, our implementation allows systems to use any desired policy for limiting the size of their logs and to fall back on an efficient incremental checkpoint transfer to transmit updates that have been garbage collected from the log. This feature both limits storage overheads and improves support for synchronizing intermittently connected devices. Second, our implementation uses self-tuning body propagation to enable prefetching policies that are simultaneously aggressive and safe. Third, our implementation provides incremental log exchange to allow systems to minimize the window for conflicting updates. Due to space constraints, we briefly outline these aspects of the implementation and provide additional details in an extended technical report [3].

Incremental checkpoint transfer. Imprecise invalidations yield an unexpected benefit: incremental checkpoint transfer.

Nodes can garbage collect any prefix of their logs, which allows each node to bound the amount local storage used for the log to any desired fraction of its total disk space. But, if a node nI garbage collects log entries older than nI.omitVV and another node n2 requests

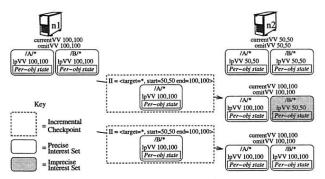


Fig. 6: Incremental checkpoints from n1 to n2.

a log exchange beginning before nI.omitVV, then nI cannot send a stream of invalidations. Instead, nI must send a checkpoint of its per-object state.

In existing server replication protocols [26], in order to ensure consistency, such a checkpoint exchange must atomically update n2's state for all objects in the system. Otherwise, the prefix property and causal consistency invariants could be violated. Traditional checkpoint exchanges, therefore, may block interactive requests while the checkpoint is atomically assembled at n1 or applied at n2, and they may waste system resources if a checkpoint transfer is started but fails to complete.

Rather than transferring information about all objects, an incremental checkpoint updates an arbitrary interest set. As Figure 6 illustrates, an incremental checkpoint for interest set IS includes (1) an imprecise invalidation that covers all objects from the receiver's currentVV up to the sender's currentVV, (2) the logical time of the sender's per-object state for IS (IS.lastPreciseVV), and (3) per-object state: the logical timestamp for each object in IS whose timestamp exceeds the receiver's IS.lastPrecise-VV. Thus, the receiver's state for IS is brought up to include the updates known to the sender, but other interest sets may become IMPRECISE to enforce consistency.

Overall, this approach makes checkpoint transfer a much smoother process than under existing protocols. As Figure 6 illustrates, the receiver can receive an incremental checkpoint for a small portion of its ID space and then either background fetch checkpoints of other interest sets or fault them in on demand.

Self-tuning body propagation. In addition to supporting demand-fetch of particular objects, our prototype provides a novel self-tuning prefetching mechanism. A node nI subscribes to updates from a node n2 by sending a list L of directories of interest along with a startVV version vector. n2 will then send n1 any bodies it sees that are in L and that are newer than startVV. To do this, n2 maintains a priority queue of pending sends: when a new eligible body arrives, n2 deletes any pending sends of older versions of the same object and then inserts a reference to the updated object. This priority queue drains

to nI via a low-priority network connection that ensures that prefetch traffic does not consume network resources that regular TCP connections could use [36]. When a lot of spare bandwidth is available, the queue drains quickly and nearly all bodies are sent as soon as they are inserted. But, when little spare bandwidth is available, the buffer sends only high priority updates and absorbs repeated writes to the same object.

Incremental log propagation. The prototype implements a novel variation on existing batch log exchange protocols. In particular, in the batch log exchange used in Bayou, a node first receives a batch of updates comprising a start time *startVV* and a series of writes, it then rolls back its checkpoint to before *startVV* using an undo log, and finally it rolls forward, merging the newly received batch of writes with its existing redo log and applying updates to the checkpoint. In contrast, our incremental log exchange applies each incoming write to the current checkpoint state without requiring roll-back and roll-forward of existing writes.

The advantages of the incremental approach are efficiency (each write is only applied to the checkpoint once), concurrency (a node can process information from multiple continuous streams), and consistency (connected nodes can stay continuously synchronized which reduces the window for conflicting writes.) The disadvantage is that it only supports simple conflict detection logic: for our incremental algorithm, a node detects a write/write conflict when an invalidation's prevAccept logical time (set by the original writer to equal the logical time of the overwritten value) differs from the logical time the invalidation overwrites in the node's checkpoint. Conversely, batch log exchange supports more flexible conflict detection: Bayou writes contain a dependency_check procedure that can read any object to determine if a conflict has occurred [34]; this approach works in a batch system because rollback takes all of the system's state to a logical moment in time at which these checks can be re-executed. Note that this variation is orthogonal to the PRACTI approach: a full replication system such as Bayou could be modified to use our incremental log propagation mechanism, and a PRACTI system could use batch log exchange with roll-back and roll-forward.

4 Evaluation

We have constructed a prototype PRACTI system written in Java and using BerkeleyDB [33] for per-node local storage. All features described in this paper are implemented including local create/read/write/delete, flexible consistency, incremental log exchange, remote read and prefetch, garbage collection of the log, incremental checkpoint transfer between nodes, and crash recovery.

We use this prototype both (1) to evaluate the PRACTI architecture in several environments such as web service replication, data access for mobile users, and grid scientific computing and (2) to characterize PRACTI's properties across a range of key metrics.

Our experiments seek to answer two questions.

- Does a PRACTI architecture offer significant advantages over existing replication architectures? We find that our system can dominate existing approaches by providing more than an order of magnitude better bandwidth and storage efficiency than AC-TI replicated server systems, as much as an order of magnitude better synchronization delay compared to PR-AC hierarchical systems, and consistency guarantees not achievable by PR-TI per-object replication systems.
- 2. What are the costs of PRACTI's generality? Given that a flexible PRACTI protocol can subsume existing approaches, is it significantly more expensive to implement a given system using PRACTI than to implement it using narrowly-focused specialized mechanisms? We find that the primary "extra" cost of PRACTI's generality is that our system can transmit more consistency information than a customized system might require. But, our implementation reduces this cost compared to past systems via separating invalidations and bodies and via imprecise invalidations, so these costs appear to be minor.

To provide a framework for exploring these issues, we first focus on partial replication in 4.1. We then examine topology independence in 4.2. Finally, we examine the costs of flexible consistency in 4.3.

4.1 Partial replication

When comparing to the full replication protocols from which our PRACTI system descends, we find that support for partial replication dramatically improves performance for three reasons:

- Locality of Reference: partial replication of bodies and invalidations can each reduce storage and bandwidth costs by an order of magnitude for nodes that care about only a subset of the system's data.
- Bytes Die Young: partial replication of bodies can significantly reduce bandwidth costs when "bytes die young" [2].
- Self-tuning Replication: self-tuning replication minimizes response time for a given bandwidth budget.

It is not a surprise that partial replication can yield significant performance advantages over existing server replication systems. What is significant is that (1) our experiments provide evidence that despite the good properties of server replication systems (e.g., support for disconnected operation, flexible consistency, and dynamic network topologies) these systems may be impractical for

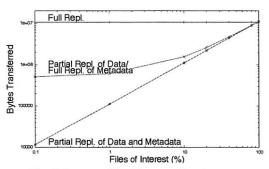


Fig. 7: Impact of locality on replication cost.

many environments; and (2) they demonstrate that these trade-offs are not fundamental—a PRACTI system can support PR while retaining the good AC-TI properties of server replication systems.

Locality of reference. Different devices in a distributed system often access different subsets of the system's data because of locality and different hardware capabilities. In such environments, some nodes may access 10%, 1%, or less of the system's data, and partial replication may yield significant improvements in both bandwidth to distribute updates and space to store data.

Figure 7 examines the impact of locality on replication cost for three systems implemented on our PRACTI core using different controllers: a full replication system similar to Bayou, a partial-body replication system that sends all precise invalidations to each node but that only sends some bodies to a node, and a partial-replication system that sends some bodies and some precise invalidations to a node but that summarizes other invalidations using imprecise invalidations. In this benchmark, we overwrite a collection of 1000 files of 10KB each. A node subscribes to invalidations and body updates for the subset of the files that are of interest to that node. The x axis shows the fraction of files that belong to a node's subset, and the y axis shows the total bandwidth required to transmit these updates to the node.

The results show that partial replication of both bodies and invalidations is crucial when nodes exhibit locality. Partial replication of bodies yields up to an order of magnitude improvement, but it is then limited by full replication of metadata. Using imprecise invalidations to provide true partial replication can gain over another order of magnitude as locality increases.

Note that Figure 7 shows bandwidth costs. Partial replication provides similar improvements for space requirements (graph omitted.)

Bytes die young. Bytes are often overwritten or deleted soon after creation [2]. Full replication systems send all writes to all servers, even if some of the writes are quickly made obsolete. In contrast, PRACTI replication can send invalidations separately from bodies: if

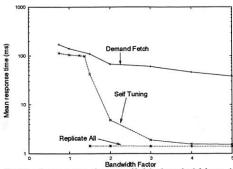


Fig. 8: Read response time available bandwidth varies for full replication, demand reads, and self-tuning replication.

a file is written multiple times on one node before being read on another, overwritten bodies need never be sent.

To examine this effect, we randomly write a set of files on one node and randomly read the files on another node. Due to space constraints, we defer the graph to the extended report [3]. To summarize: when the write to read ratio is 2, PRACTI uses 55% of the bandwidth of full replication, and when the ratio is 5, PRACTI uses 24%.

Self-tuning replication. Separation of invalidations from bodies enables a novel self-tuning data prefetching mechanism described in Section 3.4. As a result, systems can replicate bodies aggressively when network capacity is plentiful and replicate less aggressively when network capacity is scarce.

Figure 8 illustrates the benefits of this approach by evaluating three systems that replicate a web service from a single origin server to multiple edge servers. In the dissemination services we examine, all updates occur at the origin server and all client reads are processed at edge servers, which serve both static and dynamic content. We compare the read response time observed by the edge server when accessing the database to service client requests for three replication policies: Demand Fetch follows a standard client-server HTTP caching model by replicating precise invalidations to all nodes but sending new bodies only in response to demand requests, Replicate All follows a Bayou-like approach and replicates both precise invalidations and all bodies to all nodes, and Self Tuning exploits PRACTI to replicate precise invalidations to all nodes and to have all nodes subscribe for all new bodies via the self-tuning mechanism. We use a synthetic workload where the read:write ratio is 1:1, reads are Zipf distributed across files ($\alpha = 1.1$), and writes are uniformly distributed across files. We use Dummynet to vary the available network bandwidth from 0.75 to 5.0 times the system's average write throughput.

As Figure 8 shows, when spare bandwidth is available, self-tuning replication improves response time by up to a factor of 20 compared to *Demand-Fetch*. A key challenge, however, is preventing prefetching from overloading the system. Whereas our self-tuning approach adapts

	Storage	Dirty Data	Wireless	Internet
Office server	1000GB	100MB	10Mb/s	100Mb/s
Home desktop	10GB	10MB	10Mb/s	1Mb/s
Laptop	10GB	10MB	10Mb/s 1Mb/s	50Kb/s Hotel only
Palmtop	100MB	100KB	1Mb/s	N/A

Fig. 9: Configuration for mobile storage experiments.

bandwidth consumption to available resources, *Replicate All* sends all updates regardless of workload or environment. This makes *Replicate All* a poor neighbor—it consumes prefetching bandwidth corresponding to the current write rate even if other applications could make better use of the network.

4.2 Topology independence

We examine topology independence by considering two environments: a mobile data access system distributed across multiple devices and a wide-area-network file system designed to make it easy for PlanetLab and Grid researchers to run experiments that rely on distributed state. In both cases, PRACTI's combined partial replication and topology independence allows our design to dominate topology-restricted hierarchical approaches by doing two things:

- 1. Adapt to changing topologies: a PRACTI system can make use of the best paths among nodes.
- Adapt to changing workloads: a PRACTI system can optimize communication paths to, for example, use direct node-to-node transfers for some objects and distribution trees for others.

We primarily compare against standard restrictedtopology client-server systems like Coda and IMAP. For completeness, our graphs also compare against topologyindependent, full replication systems like Bayou.

Mobile storage. We first consider a mobile storage system that distributes data across palmtop, laptop, home desktop, and office server machines. We compare a PRACTI system to a client-server system that supports partial replication but that distributes updates via a central server and to a full-replication system that can distribute updates directly between any nodes but that requires full replication. All three systems are realized by implementing different controller policies.

As summarized in Figure 9 our workload models a department file system that supports mobility: an office server stores data for 100 users, a user's home machine and laptop each store one user's data, and a user's palmtop stores 1% of a user's data. Note that due to resource limitations, we store only the "dirty data" on our test machines, and we use desktop-class machines for all nodes. We control the network bandwidth of each scenario using a library that throttles transmission.

Figure 10 shows the time to synchronize dirty data among machines in three scenarios: (a) *Plane*: the user

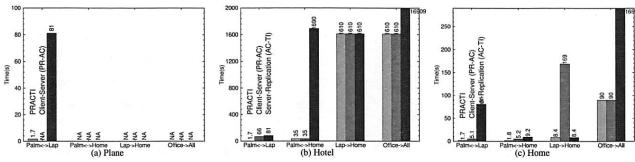


Fig. 10: Synchronization time among devices for different network topologies and protocols.

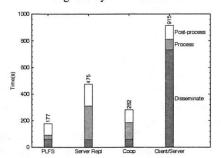


Fig. 11: Execution time for the WAN-Experiment benchmark on 50 distributed nodes with a remote server.

is on a plane with no Internet connection, (b) *Hotel*: the user's laptop has a 50Kb/s modem connection to the Internet, and (c) *Home*: the user's home machine has a 1Mb/s connection to the Internet. The user carries her laptop and palmtop to each of these locations and colocated machines communicate via wireless network at speeds indicated in Figure 9. For each location, we measure time for machines to exchange updates: (1) $P \leftrightarrow L$: the palmtop and laptop exchange updates, (2) $P \leftrightarrow H$: the palmtop and home machine exchange updates, (3) $L \rightarrow H$: the laptop sends updates to the home machine, (4) $O \rightarrow All$: the office server sends updates to all nodes.

In comparing the PRACTI system to a client-server system, topology independence has significant gains when the machines that need to synchronize are near one another but far from the server: in the isolated *Plane* location, the palmtop and laptop can not synchronize at all in a client-server system; in the *Hotel* location, direct synchronization between these two co-located devices is an order of magnitude faster than synchronizing via the server (1.7s v. 66s); and in the *Home* location, directly synchronizing co-located devices is between 3 and 20 times faster than synchronization via the server.

WAN-FS for Researchers. Figures 11 and 12 evaluate a wide-area-network file system called PLFS designed for PlanetLab and Grid researchers. The controller for PLFS is simple: for invalidations, PLFS forms a multicast tree to distribute all precise invalidations to all nodes. And, when an *INVALID* file is read, PLFS uses a DHT-based system [38] to find the nearest copy of the

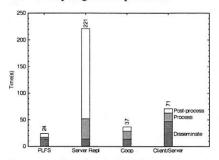


Fig. 12: Execution time for the WAN-Experiment benchmark on 50 cluster nodes plus a remote server.

file; not only does this approach minimize transfer latency, it effectively forms a multicast tree when multiple concurrent reads of a file occur [1, 35].

We examine a 3-phase benchmark that represents running an experiment: in phase 1 *Disseminate*, each node fetches 10MB of new executables and input data from the user's home node; in phase 2 *Process*, each node writes 10 files each of 100KB and then reads 10 files from randomly selected peers; in phase 3, *Post-process*, each node writes a 1MB output file and the home node reads all of these output files. We compare PLFS to three systems: a client-server system, client-server with cooperative caching of read-only data [1], and server-replication [26]. All 4 systems are implemented via PRACTI using different controllers.

The figures show performance for an experiment running on 50 distributed nodes each with a 5.6Mb/s connection to the Internet (we emulate this case by throttling bandwidth) and 50 cluster nodes at the University of Texas with a switched 100Mb/s network among them and a shared path via Internet2 to the origin server at the University of Utah.

The speedups range from 1.5 to 9.2, demonstrating the significant advantages enabled by the PRACTI architecture. Compared to client/server, it is faster in both the Dissemination and Process phases due to its multicast dissemination and direct peer-to-peer data transfer. Compared to full replication, it is faster in the Process and Post-process phases because it only sends the required data. And compared to cooperative caching of read only data, it is faster in the Process phase because data is trans-

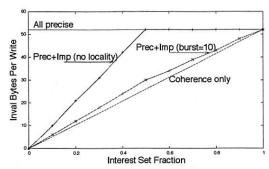


Fig. 13: Bandwidth cost of consistency information.

ferred directly between nodes.

4.3 Arbitrary consistency

This subsection examines the costs of PRACTI's generality. As Section 3.3 describes, our protocol ensures that requests pay only the latency and availability costs of the consistency they require. But, distributing sufficient bookkeeping information to support a wide range of per-request semantics does impose a bandwidth cost. If all applications in a system only care about coherence guarantees, a customized protocol for that system could omit imprecise invalidations and thereby reduce network overheads.

Three features of our protocol minimize this cost. First, transmitting invalidations separately from bodies allows nodes to maintain a consistent view of data without receiving all bodies. Second, transmitting imprecise invalidations in place of some precise invalidations allows nodes to maintain a consistent view of data without receiving all precise invalidations. Third, self-tuning prefetch of bodies allows a node to maximize the amount of local, valid data in a checkpoint for a given bandwidth budget. In an extended technical report [3], we quantify how these features can greatly reduce the cost of enforcing a given level of consistency compared to existing server replication protocols.

Figure 13 quantifies the remaining cost to distribute both precise and imprecise invalidations (in order to support consistency) versus the cost to distribute only precise invalidations for the subset of data of interest and omitting the imprecise invalidations (and thus only supporting coherence.) We vary the fraction of data of interest to a node on the x axis and show the invalidation bytes received per write on the y axis. Our workload is a series of writes by remote nodes in which all objects are equally likely to be written. Note that the cost of imprecise invalidations depends on the workload's locality: if there is no locality and writers tend to alternate between writing objects of interest and objects not of interest, then the imprecise invalidations between the precise invalidations will cover relatively few updates and save relatively little overhead. Conversely, if writes to different interest sets arrive in bursts, then the system will generally be able to accumulate large numbers of updates into imprecise invalidations. We show two cases: the *No Locality* line shows the worst case scenario, with no locality across writes, and the *burst=10* line shows the case when a write is ten times more likely to hit the same interest set as the previous write than to hit a new interest set.

When there is significant locality for writes, the cost of distributing imprecise invalidations is small: imprecise invalidations to support consistency never add more than 20% to the bandwidth cost of supporting only coherence. When there is no locality, the cost is higher, but in the worst case imprecise invalidations add under 50 bytes per precise invalidation received. Overall, the difference in invalidation cost is likely to be small relative to the total bandwidth consumed by the system to distribute bodies.

5 Related work

Replication is fundamentally difficult. As noted in Section 3.3, the CAP dilemma [9, 31] and performance/consistency dilemma [22] describe fundamental trade-offs. As a result, systems *must* make compromises or optimize for specific workloads. Unfortunately, these workload-specific compromises are often reflected in system mechanisms, not just their policies.

In particular, state of the art mechanisms allow a designer to retain full flexibility along at most two of the three dimensions of replication, consistency, or topology policy. Section 2 examines existing PR-AC [15, 18, 25], AC-TI [10, 17, 19, 26, 37, 39], and PR-TI [11, 29] approaches. These systems can be seen as special case "projections" of the more general PRACTI mechanisms.

Some recent work extends server replication systems towards supporting partial replication. Holliday et al.'s protocol allows nodes to store subsets of data but still requires all nodes to receive updates for all objects [14]. Published descriptions of Shapiro et al.'s consistency constraint framework focus on full replication, but the authors have sketched an approach for generalizing the algorithms to support partial replication [30].

Like PRACTI, the Deceit file system [31] provides a flexible substrate that subsumes a range of replication systems. Deceit, however, focuses on replication across a handful of well-connected servers, and it therefore makes very different design decisions than PRACTI. For example, each Deceit server maintains a list of all files and of all nodes replicating each file, and all nodes replicating a file receive all bodies for all writes to the file.

A description of PRACTI was first published in an earlier technical report [7], and an extended technical report [3] describes the current system in more detail.

6 Conclusion

In this paper, we introduce the PRACTI paradigm for replication in large scale systems and we describe the first system to simultaneously provide all three PRACTI properties. Evaluation of our prototype suggests that by disentangling mechanism from policy, PRACTI replication enables significantly better trade-offs for system designers than possible with existing mechanisms. By subsuming existing approaches and enabling new ones, we speculate that PRACTI may serve as the basis for a unified replication architecture that simplifies the design and deployment of large-scale replication systems.

Acknowlegements

We thank Lorenzo Alvisi, Eric Brewer, John Byers, Allen Clement, Ravi Kokku, Jean-Phillipe Martin, Jayaram Mudigonda, Lili Qiu, Marc Shapiro, Harrick Vin, Emmett Witchel, and Haifeng Yu for their feedback on earlier drafts of this paper.

References

- S. Annapureddy, M. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc NSDI*, May 2005.
- [2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proc. ASPLOS*, pages 10–22, Sept. 1992.
- [3] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems (extended version). Technical Report UTCS-06-17, U. of Texas at Austin, Apr. 2006.
- [4] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In ICDCS, June 1992.
- [5] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In USENIX Conf. on Domain-Specific Lang., Oct. 1997.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In SOSP, Oct. 2001.
- [7] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication for large-scale systems. Technical Report TR-04-28, U. of Texas at Austin, Mar. 2004.
- [8] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In Wkshp. on Internet Svr. Perf., June 1998.
- [9] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In ACM SIGACT News, 33(2), Jun 2002.
- [10] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4):379–405, 1992.
- [11] R. Guy, J. Heidemann, W. Mak, T. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer Conf.*, pages 63–71, June 1990.
- [12] J. Hennessy and D. Patterson. Computer Architecture A Quantitative Approach. Morgan Kaufmann, Inc., 2nd edition, 1996.
- [13] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. ACM Trans. Prog. Lang. Sys., 12(3), 1990.
- [14] J. Holliday, D. Agrawal, and A. E. Abbadi. Partial database replication using epidemic communication. In *ICDCS*, July 2002.
- [15] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. ACM Trans. on Computer Systems, 6(1):51– 81, Feb. 1988.
- [16] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *ICDCS*, pages 302–311, 1990.
- [17] P. Keleher. Decentralized replicated-object protocols. In PODC, pages 143–151, 1999.
- [18] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. ACM Trans. on Computer Systems, 10(1):3– 25, Feb. 1992.

- [19] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. ACM Trans. on Computer Systems, 10(4):360–391, 1992.
- [20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Comm. of the ACM, 21(7), July 1978.
- [21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
- [22] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.
- [23] D. Malkhi and D. Terry. Concise version vectors in WinFS. In Symp. on Distr. Comp. (DISC), 2005.
- [24] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In USENIX Winter Conf., pages 305–313, Jan. 1992.
- [25] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. ACM Trans. on Computer Systems, 6(1), Feb. 1988.
- [26] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In SOSP, Oct. 1997.
- [27] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. USENIX FAST*, Mar. 2003.
- [28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In SOSP, 2001.
- [29] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. OSDI*, Dec. 2002.
- [30] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In Proc. OPODIS. Dec. 2004.
- [31] A. Siegel. Performance in Flexible Distributed File Systems. PhD thesis, Cornell. 1992.
- [32] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In SPAA, 1997.
- [33] Sleepycat Software. Getting Started with BerkeleyDB for Java, Sept. 2004.
- [34] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In SOSP, Dec. 1995.
- [35] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS*, May 1999.
- [36] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In Proc. OSDI, Dec. 2002.
- [37] G. Wuu and A. Berstein. Efficient solutions to the replicated log and dictionary problem. In PODC, pages 233–242, 1984.
- [38] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.
- [39] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. on Computer Systems, 20(3), Aug. 2002.

Exploiting Availability Prediction in Distributed Systems

James W. Mickens and Brian D. Noble
EECS Department, University of Michigan
Ann Arbor, MI, 48103
{imickens,bnoble}@umich.edu

Abstract

Loosely-coupled distributed systems have significant scale and cost advantages over more traditional architectures, but the availability of the nodes in these systems varies widely. Availability modeling is crucial for predicting per-machine resource burdens and understanding emergent, system-wide phenomena. We present new techniques for predicting availability and test them using traces taken from three distributed systems. We then describe three applications of availability prediction. The first, availability-guided replica placement, reduces object copying in a distributed data store while increasing data availability. The second shows how availability prediction can improve routing in delay-tolerant networks. The third combines availability prediction with virus modeling to improve forecasts of global infection dynamics.

1 Introduction

Cooperative, opt-in distributed systems provide attractive benefits, but face at least one significant challenge: nodes can enter and leave the collective on a whim, because each machine may be separately owned and managed [3, 6, 29]. Such *churn* can lead to significant overheads [5]. However, if one could predict the availability of even a portion of the nodes with reasonable accuracy, one could reduce these costs by planning for changing availability instead of reacting to it.

This paper introduces new techniques for predicting node availability. Our predictors are less conservative than previous analytical models [2, 5] and more accurately capture phase relationships between the availability of different nodes [30]. We test our predictors using availability traces from the PlanetLab distributed test bed [33], the Microsoft Corporation [6], and the Overnet distributed hash table [3]. Each set of machines has a distinct predictability profile, and we explain the differences in predictability by uncovering the generic classes of uptime patterns found in each data set.

We then provide three applications of our prediction techniques. First, we show how to reduce object copying in a distributed hash table by biasing replicas towards highly available nodes. We can eliminate the majority of non-optimal

copies incurred by the standard replication scheme while increasing data availability by at least a factor of two.

We then apply our predictors to two delay-tolerant networking scenarios [13], using them to approximate the role of *contact oracles* [16]. Under reasonable load factors, our predictive schemes reduce delivery latency to within 15% of the latency provided by an oracle.

Finally, we show how explicit representations of machine availability can improve the fidelity of epidemic spreading models [17, 25, 34]. These models typically assume that nodes are always online, so they over-estimate infection levels. By incorporating the availability fluctuations found in real systems, these models can capture diurnal variations in the spreading rate and forecast steady state infection levels that are much closer to those actually exhibited.

2 Related Work

There are many empirical studies of availability in distributed systems. Most used active network probing to detect uptime changes. Bolosky et al described the uptimes of over 50,000 PCs belonging to the Microsoft Corporation [6]. Saroiu et al studied Napster and Gnutella, popular peer-to-peer file trading services [29]. Douceur performed a meta-analysis of availability data [8], examining the Microsoft, Gnutella, and Napster traces, as well as a trace from a sampling of global Internet hosts [20]. Douceur posited two broad classes of machine availability; those in the first are almost always online, whereas those in the second have diurnal uptime periods. Bhagwan et al studied nodes in the Overnet DHT and also found diurnal uptime patterns [3]. We extend these binary categorizations by providing a richer taxonomy of uptime classes and automatic methods for identifying these classes in availability traces.

Rather than rely on coarse-grained probing, other studies have used operating system logs to infer downtime. For example, Simache's analysis of a Unix workstation cluster found a median downtime of 38.5 minutes [31]; roughly 35% of reboots were caused by 10% of the machines, primarily between 8AM and 6PM.

Analytic models of cooperative systems typically include a parameter for host availability. Douceur and Wattenhofer expressed a machine's availability in terms of its fractional downtime [10]. To account for time-of-day effects on global aggregate availability, Bhagwan *et al* assumed a pessimistic mean availability based only on hosts online at night [2]. Blake and Rodrigues also used conservative, average-based metrics in their model [5].

Such conservative estimates are useful in provisioning against worst-case scenarios, but they cannot predict the evolution of the system over time, and they cannot predict individual node behavior well if a system contains heterogeneous availability patterns. To do these things, we need to estimate individual or aggregate availability at multiple points in the future; our new schemes do this. However, we do not investigate the long term admission/drop-out rate, which is also a major issue in peer-to-peer environments [3].

In the computational grid community, availability prediction is done by fitting empirically observed uptime traces to well-known statistical distributions such as the Pareto or Weibull distribution [24]. Using the derived model parameters, one can estimate how long a random machine will be online before failing. This approach suffers from limitations similar to those of the conservative estimates.

Several cooperative systems have been designed to deal with varying host availability. Total Recall is a peer-to-peer storage system which adjusts replication strategies to meet specified data availability goals [4]. It adapts to changing file workloads and node churn, providing two methods for maintaining data redundancy. In eager repair, the system reacts to a host going down by replicating its data elsewhere. With lazy repair, the system estimates worst-case host availability using Bhagwan's conservative estimates [2]; it then overreplicates data such that redundancy targets are still met in these worst case scenarios. Lazy repair trades increased disk requirements for reduced bandwidth. Using our availability predictors, we can reduce both bandwidth and disk consumption. By identifying nodes that are highly available and biasing data storage towards them, we decrease on-demand object regeneration while reducing the need for over-replication.

Mahajan et al showed how a node in a peer-to-peer system can estimate the mean session time by observing the churn rate in its neighbor set [21]. Machines can then reduce the rate at which they send keep-alive messages while maintaining the same level of consistency in their routing tables, reducing control traffic. Knowing the mean session time of the network at a particular moment cannot be used to predict the availability of individual machines in a fine-grained manner.

Schwarz *et al* proposed a distributed object store which biases data storage towards peers with high predicted availability [30]. Each node has a counter which is initialized to 0. During a periodic system-wide scan, a node's counter is incremented by 1 if it is online, otherwise the counter is decremented by 1. Data storage is biased towards nodes with high

counter values. Such a system will correctly identify consistently online nodes as good storage hosts and consistently offline hosts as poor replica sites. However, the counter mechanism cannot consistently capture phase relationships within and between nodes, e.g., the fact that if a host has diurnal availability, then it will be online for the longest consecutive stretch starting in the morning, when its counter is lowest. In Section 5.1, we describe a data store which handles this correctly.

3 Availability Predictors

In this section, we present our models for availability prediction. Each individual predictor is presented, followed by a mechanism to combine them to the best possible advantage.

3.1 Saturating Counter Predictors

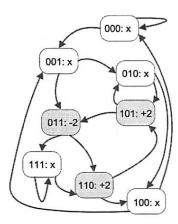
Our first predictor is the *RightNow* predictor. A node's current online status is used as the value of all predictions for all lookahead periods. RightNow predictors are attractive because they require only one bit of state, and they should work well for nodes which are predominantly online or predominantly offline. Unfortunately, they cannot accurately capture periodic availability patterns over medium or long term time scales.

We can generalize the RightNow predictor to utilize n bits of state. For example, whereas the RightNow predictor represents uptime state using a single bit, the SatCount-2 predictor uses a 2-bit saturating counter. Such a counter can assume four values (-2,-1,+1,and +2) which correspond to four uptime states (strongly offline, weakly offline, weakly online, and strongly online). During each sampling period, the counter is incremented if the node is online, otherwise it is decremented; increments and decrements cannot move the counter beyond the saturated counts of -2 and +2. Predictions for all lookahead periods use the current value of the saturating counter, i.e., negative counter values produce "offline" predictions, whereas positive values result in "online" predictions.

By using a few extra bits of storage, SatCount-x predictors are more tolerant than RightNow predictors to occasional deviations from long stretches of uniform uptime behavior. However, like the RightNow predictors, they are inaccurate for nodes with periodic uptimes.

3.2 State-Based Predictors

To predict the behavior of nodes with periodic availabilities, we turn to state-based predictors. These predictors explicitly represent a node's recent uptime states using a de Bruijn graph. A de Bruijn graph over k bits has a vertex for each binary number in $[0, 2^k-1]$. Each vertex with binary label $b_1b_2...b_k$ has two outgoing edges, one to the node labeled $b_2b_3...0$ and the other to the node $b_2b_3...1$. In other words, the transition from a starting node to a child node represents a left shift of the parent's label and an addition of 0 or 1.



This is the de Bruijn graph for uptime pattern {110}*. Each labeled vertex as uptime_state:sat_counter_value. counter A value of x means that the associated vertex has never been visited. Traversing an edge represents a left shift-and-fill of the starting node label.

Figure 1: History predictor example

Suppose that we represent a node's recent availability as a k-bit binary string, with b_i equal to 0 if the node was offline during the i^{th} most recent sampling period and 1 if it was online. A k-bit de Bruijn graph will represent each possible transition between availability states. To assist uptime predictions, we attach a 2-bit saturating counter to each vertex. These counters represent the likelihood of traversing a particular outbound edge; negative counter values bias towards the 0 path, whereas positive values bias towards the 1 path. After each uptime sampling, the counter for the vertex representing the previous uptime state is incremented or decremented according to whether the new uptime sample represented an "online" edge or an "offline" edge.

To make an uptime prediction for n time steps into the future, we trace the most likely path of n edges starting from the vertex representing the current uptime state. If the last bit we shift in is a 1, we predict the node will be online in n time units, otherwise we predict that it will be offline.

Figure 1 depicts the state maintained by such a *History* predictor. In this example, the node's availability has a periodicity of 3 samples and the repeated uptime string is 110. The node does not deviate from this pattern, so only the three shaded vertices represent observable uptime states.

The de Bruijn graphs used by the History predictor resemble the file access trees used by certain cache prefetching algorithms [14, 19]. However, these algorithms weigh each edge using raw access counts instead of saturating counters. CPU branch predictors [22] associate saturating counters with previously observed branch histories, but they do not use the superposition idea described below.

3.3 Tolerating Noise in the State Space

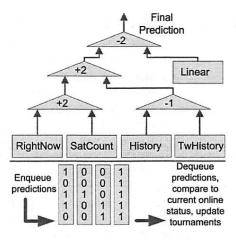
Suppose that a node has a fundamentally cyclical uptime pattern, but the pattern is "noisy." For example, a machine might be online 80% of the time between midnight and noon and always offline at other times. If the punctuated downtime between midnight and noon is randomly scattered, the de Bruijn graph will accumulate infrequently visited vertices whose labels contain mostly 1's but differ in a small number of bit positions. As the length of time that we observe the node grows, noisy downtime will generate increasingly more vertices whose labels are within a few bit-flips of each other. Probabilistically speaking, we should always predict that the node will be online from midnight to noon. However, the many vertices representing this time interval are infrequently visited and thus infrequently updated. Their counters may have weak saturations (-1 or +1) that poorly capture the underlying cyclic availability.

For nodes like this, we can nudge predictions towards the probabilistically favored ones by considering *superpositions* of multiple uptime states. Given a vertex v representing the current uptime history, we make a prediction by considering v's counter and the counters of all observed vertices whose labels differ from v's by at most d bits. For example, suppose that k=3 and d=1, and that each of the 2^k = 8 possible vertices corresponds to an actually observed uptime history. To make a prediction for the next time step when the current vertex has the label 111, we average the counter values belonging to vertices 111, 110, 101, and 011. If the average is greater than 0, we predict "online," otherwise we predict "offline."

We call such a predictor a TwiddledHistory predictor, since it considers the current vertex and all "twiddled" vertices whose labels differ by up to d bits. The hope is that by averaging the counters of similar uptime states, we remove noise and discover stable underlying availability patterns. The TwiddledHistory strategy will perform worse than the regular History strategy when vertices within d bits of each other correspond to truly distinct uptime patterns. In these situations, superposition amalgamates state belonging to unrelated availability behavior, reducing prediction accuracy.

3.4 Linear Predictor

Linear prediction [15] is a common technique from digital signal processing and statistical time series analysis. It uses a linear combination of the last k signal points to predict future points. The k coefficients are chosen to reduce the magnitude of an error signal, which is assumed to be uncorrelated with the underlying "pure" signal. To make availability predictions for t time steps into the future, we iteratively evaluate the linear combination using the k most recent availability samples, shifting out the oldest data point and shifting in the predicted data point. Linear prediction produces good estimates for signals that are stable in the short term but oscillatory in the medium to long term [27]. We would expect this



This figure depicts the tournament counters and update queue of a Hybrid predictor. The five bits in each queue entry represent the previous predictions of the five sub-predictors.

Figure 2: Hybrid predictor example

technique to work well with nodes having diurnal uptimes, e.g., machines that are online during the work day and offline otherwise.

3.5 Hybrid Predictor

A machine can transition between multiple availability patterns during its lifetime. Furthermore, some availability patterns are best modeled using different predictors for different lookahead periods. To dynamically select the best predictor for a given uptime pattern and lookahead interval, we employ a Hybrid predictor. Our approach is similar in spirit to the "mixture of experts" strategy of the Network Weather Service [35], but closer in design to hybrid branch predictors [22]. For each lookahead period of interest, the Hybrid predictor maintains tournament counters. These saturating counters determine the best predictor to use for that lookahead period. For example, Figure 2 depicts a three-level tournament. Negative counter values select the left input, whereas positive values select the right. In this example, the SatCount predictor is currently more accurate than the RightNow predictor. Similarly, the History strategy is outperforming the TwiddledHistory strategy. The best history-based approach is beating the best "simple" approach, and the Linear predictor performs worse than the best of the other four predictors. Thus, the final output of the Hybrid predictor is the History prediction.

Consider a Hybrid predictor making forecasts for an n-sample lookahead period. At the beginning of each time unit, the Hybrid predictor samples the current uptime state of its node. Its five sub-predictors are updated with this state, and each sub-predictor makes a prediction for n time units into the future. The final output of the Hybrid predictor is selected via tournaments as shown in Figure 2, and the individual sub-predictions are placed in a queue and timestamped

with $curr_time + n$. If the head of the queue contains an entry whose timestamp matches the current time, the entry is dequeued and the tournament counters are updated using the dequeued predictions. A tournament counter remains unchanged if both of the relevant dequeued predictions match the current uptime state or both do not match. Otherwise, one prediction was right and the other was wrong, and the tournament counter is incremented or decremented appropriately. In the last stage of the update, the $curr_time$ value is incremented.

A Hybrid predictor responsible for multiple lookahead periods keeps a separate update queue and tournament counter set for each period. Each queue and counter set is maintained using the algorithm described above.

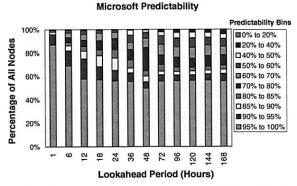
4 Predicting Individual Node Availability

We first evaluate our predictors using the PlanetLab and Microsoft availability traces. Extending a Fourier transform technique [8], we create a taxonomy of uptime classes and explain the different availability behaviors of the two traces by examining their constituent uptime classes. We then test our predictors against the Overnet availability trace and show that our models do not fare as well. We provide evidence to suggest that this is not an artifact of our predictors, but instead the result of a fundamental unpredictability in the nodes themselves.

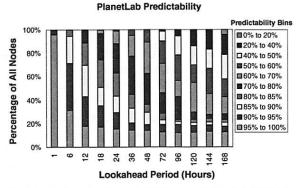
4.1 PlanetLab and Microsoft Nodes

To determine the real-world applicability of our predictors, we tested them on two empirically gathered availability traces. The first trace followed 51,662 PCs in the Microsoft corporate network [6], and the second captured the behavior of 321 nodes in the PlanetLab distributed testbed [1]. Each machine in the Microsoft trace was pinged hourly. In the PlanetLab traces [33], machines were pinged every 15 minutes, but we sampled every fourth measurement to provide a fair comparison with the Microsoft data. The lifetimes of PlanetLab nodes were long enough that such sampling did not distort the underlying availability patterns. Our PlanetLab data spanned the five week period from July 1 to August 4, 2004. The Microsoft data spanned the five week period from July 6 to August 9, 1999.

For each availability trace, we associated a unique Hybrid predictor with each node. We used the first two weeks of a node's uptime data to train its predictor. Two weeks was a reasonable training length because it gave a predictor two chances to observe uptime patterns with a periodicity of a full week. After training the predictors, we evaluated their accuracy by comparing their predictions to the remaining three weeks of availability data. During each hour in the evaluation period, the predictors made forecasts for multiple lookahead intervals and were updated with uptime samples from that hour. We say that a node is *p*-predictable for a certain looka-



(a) A stable core of Microsoft nodes remains highly predictable across all lookahead periods.



(b) PlanetLab nodes start out highly predictable, but their predictability quickly degrades. Overall, these nodes are less predictable than the Microsoft set.

Figure 3: Microsoft and PlanetLab predictability

head period if we predicted its uptime behavior with at least accuracy p.

All Hybrid predictors used 3-bit saturating tournament counters. The organization of the tournaments resembled the structure shown in Figure 2. However, instead of a single Linear predictor, two Linear predictors competed with each other—one tracked 168 bits of state and the other tracked 336 bits. Also, the History and TwiddledHistory predictors were replaced with two two-level tournaments, allowing the same predictor type with different k values to compete. The single History predictor was replaced with a two-level tournament comparing k values of 6, 24, 48, and 56; the same k values competed in the TwiddledHistory multi-level tournament. In all experiments, the SatCount predictors used 2 bits of uptime state, and the TwiddledHistory predictors used a d of 1.

Figure 3 bins the predictability of individual Microsoft and PlanetLab nodes for several lookahead intervals. For a 1-hour lookahead period, 95.6% of PlanetLab nodes can be predicted with greater than 95% accuracy, as compared to only 87.0% of Microsoft nodes. However, as the lookahead period increases, the percentage of PlanetLab nodes that are 95%-predictable quickly drops below 20%. In contrast, slightly over half of the Microsoft nodes are 95%-predictable across all lookahead intervals.

Node Type	Microsoft	PlanetLab	
Always On	60.66%	15.58%	
Always Off	1.22%	5.60%	
WW Periodic	9.79%	0.00%	
Long Stretch	20.48%	67.60%	
Unstable 70-90	2.05%	1.25%	
Unstable 50-70	1.67%	1.56%	
Unstable 10-50	4.12%	8.41%	

Figure 4: Uptime Class Categorization

As the lookahead period increases, the predictability decay of the Microsoft nodes is more graceful than that of the PlanetLab nodes. For example, with a 144 hour lookahead period, 72.5% of the PlanetLab nodes have worse than 70% predictability; in the Microsoft data set, only 22.6% of nodes are this bad. In fact, for all of the studied lookahead periods, no more than a fourth of the Microsoft nodes are ever worse than 70%-predictable.

Why do the two data sets have different predictabilities? To answer this question, we must determine the distinct uptime classes contained in each set. We extract these classes by extending a technique proposed by Douceur [8]. Douceur wanted to identify nodes with diurnal uptime patterns, e.g., machines that were online from 9 AM to 5 PM during the work week. Douceur treated each node's uptime trace as a digital signal where bit n was a 1 if the node was online during hour n. He then applied a Fourier decomposition [15] to each signal, determining the set of sine waves whose sum equaled the original signal. Nodes with diurnal uptime patterns had spikes in the daily and weekly frequency spectra.

We can generalize this technique to detect multiple types of uptime regimes. To classify a node's availability behavior, we convert its uptime string into a digital signal and apply several tests to it. Once an uptime signal passes a test, we consider it categorized and we do not apply the remaining tests.

First, we classify a node as *always on* or *always off* if it's availability signal contains 90% ones or zeros, respectively. Second, nodes are subjected to Douceur's technique to detect diurnal periodicity. Nodes that pass this test are *work-week periodic*. Third, if the Fourier decomposition resembles the curve 1/f, the node's uptime pattern is the summation of low frequency sine waves. This means that the node's online and offline stretches are long running, and we label these *long stretch nodes*. While it is possible that unstable nodes have spikes in frequency domains other than those explored, we have not found such spikes in our traces. Therefore, a node failing all of these tests is labeled *unstable*. We further bin unstable nodes according to the percentage of time that they are online. This creates the uptime classes *unstable70to90*, *unstable50to70*, and *unstable10to50*.

Figure 4 describes the constituent uptime classes in the PlanetLab and Microsoft traces. We see that PlanetLab is dominated by long stretch nodes. Long stretch nodes are highly predictable in the short term—given the current uptime

state of a long stretch node, we can confidently predict that it will remain in that state for the next few hours. However, long stretch nodes are increasingly unpredictable for larger lookahead intervals because their uptime regimes lack periodicity. Once such a node changes uptime state, it will keep that state for many hours, but the arrival of these changes are random. Thus, the predictability of the PlanetLab system as a whole degrades for long lookahead periods.

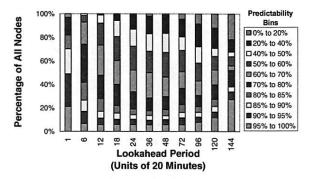
In the Microsoft data set, 61% of the machines are always on. These nodes represent the stable core which is 95%-predictable across all lookahead intervals. Whereas the PlanetLab system has no work-week periodic nodes, 9.79% of the Microsoft machines have these diurnal uptime patterns ¹. These machines are often highly predictable, although this is not always the case. For example, some machines are work-week periodic only to the extent that they are always offline during non-work day hours; during the actual work day, these machines have highly variable availability and thus are difficult to predict during these times. As another example, some work-week periodic machines are occasionally left online for multiple work days or left online during the weekend. Such aperiodic behavior is also difficult to forecast.

4.2 Predictability of Overnet Nodes

Microsoft and PlanetLab machines have fairly long session times; a node that has just come online will likely stay online for multiple consecutive hours. Nodes in other peer-to-peer networks like Gnutella or Napster have much higher churn rates [3, 29], typically on the order of tens of minutes. A natural question is whether these uptime patterns can be modeled using our prediction techniques.

To answer this question, we fed our predictors an availability trace of the Overnet DHT [3]. The trace covered 7 days with a sampling period of 20 minutes, providing 504 sample points. The trace contained 1,468 nodes that responded to at least one probe from January 15 to January 21, 2003. To evaluate the predictors on the Microsoft and PlanetLab data, we trained them for 2 weeks, which at a sample rate of once an hour resulted in 336 training samples and 504 evaluation samples. For a fair comparison, we also trained our Overnet predictors for 336 samples, leaving only 168 samples for evaluation purposes. We did not selectively pick hourly samples as we did for the PlanetLab traces because a probing granularity of 20 minutes is appropriate for a network with high churn rates. However, we did filter out nodes that were not online at least once in the first 100 samples and the last 100 sampling periods. This created a more challenging prediction

Overnet Predictability, Restricted Trace



Our availability predictions are less accurate for the Overnet trace than for the Microsoft and PlanetLab data sets.

Figure 5: Overnet predictability

environment, since many Overnet nodes were almost always offline and thus easy to predict. Bhagwan also estimated a non-trivial attrition rate of 32 hosts per day in the full node set [3]. Our paper is not concerned with long-term attrition effects, so the smaller trace set (haphazardly) filters out some of these "permanently" lost nodes.

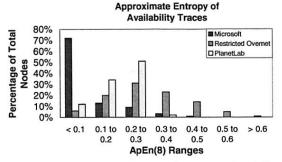
In the full Overnet trace, about a third of all nodes are 95% predictable for an arbitrary lookahead period; these are primarily nodes which are almost always off. As depicted in Figure 5, less than a tenth of the nodes in the restricted trace are 95%-predictable for an arbitrary lookahead period. In fact, the overall predictability of the restricted Overnet trace is much worse than that of the Microsoft or PlanetLab system. A natural question arises: what properties of the Overnet data set make it less predictable?

4.3 Entropy and Predictability

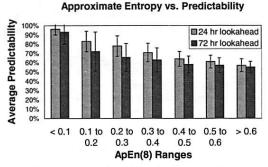
We must interpret the Overnet results from Section 4.2 with caution, since we have fewer evaluation samples than in the PlanetLab/Microsoft experiments and we cannot fully control for long term node attrition. We also do not have enough samples to confidently categorize nodes as work-week periodic, unstable, etc. However, manual inspection of the three uptime traces reveals qualitative differences in availability patterns. Overnet nodes appear to have more long stretch downtime than Microsoft or PlanetLab nodes, but the online stretches of Overnet nodes seem more randomly punctuated by bursts of downtime. This implies that Overnet nodes with non-trivial amounts of uptime should be more difficult to predict than Microsoft or PlanetLab nodes with similar uptime percentages.

To quantify this intuition, we use the information theoretic concept of approximate entropy [26], denoted ApEn(x). Given an arbitrary length input string and an integer m, ApEn(m) represents the additional information provided by the last symbol of an m-character substring, given that we already know the first m-1 characters. Approximate entropy is

¹Note that Douceur reported that 14% of Microsoft nodes have cyclical availability patterns [8], whereas we say that only 9.79% of them are workweek periodic. We report a lower percentage because we use a higher energy cutoff in the daily and weekly spectra for a node to classify as work-week periodic. From the perspective of evaluating our predictors, this more stringent cutoff is reasonable. For example, a node which is always offline except from noon to 2 PM during the work-week looks more like an always off node to our predictors. Thus, we categorize it as such.



(a) This chart categorizes the approximate entropies of the availability strings in each trace set.



(b) This graph shows the relationship between the entropy of a Microsoft availability string and its predictability. Each y-bar represents a standard deviation.

Figure 6: Approximate Entropy Results

highest when all m-character substrings have equal frequencies. When ApEn (m) is low, we conclude that the string has repeated patterns and is non-random. As a simple example, consider the string $\{10\}^*$. Knowing the first bit of a two bit substring allows perfect prediction of the following bit. Thus, ApEn (2) is close to 0.

Figure 6(a) bins the approximate entropies of the uptime strings in the Microsoft, PlanetLab, and restricted Overnet traces. This figure validates our intuition that Overnet hosts have less regular availability patterns. The ApEn (8) values of the Microsoft and PlanetLab nodes are primarily smaller than 0.3, but 42% of Overnet nodes have an ApEn (8) greater than 0.3. Figure 6(b) plots ApEn (8) versus predictability for the Microsoft data set, confirming that higher entropy values are indeed correlated with lower predictability.

Note that the PlanetLab trace has higher entropy than the Microsoft trace. Referring to Figure 4, we see that the PlanetLab system has twice as many unstable 10to 50 nodes, which we would expect to be quite random. More importantly, PlanetLab is dominated by long stretch nodes instead of always on nodes. This should also increase system entropy, since long stretch nodes have less regularity than always on machines.

Machines with high uptime entropy may be difficult to predict individually, but a possible salvation may lie in the ability to identify nodes with correlated uptimes. Nodes displaying erratic behavior when considered singly may show emergent periodic behavior when considered in aggregate. This notion is supported by the Overnet trace, which shows diurnal periodicity at the global scale. By expanding the notion of superposition to include clusters of machines, we may be able to diminish the impact of entropy upon prediction accuracy for single nodes. We are currently investigating such methods.

4.4 Discussion

The Microsoft, PlanetLab, and Overnet traces were collected using a centralized probing infrastructure. For reasons of scalability or trust, such an architecture may be undesirable in some deployment scenarios. When we discuss our availability-aware applications in Section 5, we describe several mechanisms for decentralized dissemination of availability data. Dealing with malicious hosts seems to be a more difficult problem. It is not immediately obvious how a host can prove that its actual uptime history is equivalent to one gathered through centralized pinging or self-reporting. Developing threat models for availability-aware systems is an important area of future research.

Network partitions or outages may cause gaps in availability histories. For example, if histories are compiled via centralized probing, then probes may be dropped in a correlated, system-wide manner. Alternatively, if histories are self-maintained, network outages will hamper attempts to disseminate these records to peers. We are currently developing middleware to collect and distribute availability data in the face of such events.

The Microsoft and PlanetLab traces used pings to determine availability. If machines receive IP addresses in a non-static way, e.g. from DHCP or NAT, and the probing infrastructure is unaware of such dynamic assignments, then ping-based probing can lead to an overestimation of the number of hosts and an underestimation of host availability [3]. Fortunately, IP aliasing should be rare in both of these traces. The Microsoft study performed name lookups before each round of pinging so that availability strings could be assigned to specific machines instead of specific IP addresses. Furthermore, as stated in the instruction manual for PlanetLab administrators, the primary IP address for each PlanetLab node should be a static one.

Aliasing is not a problem if per-host operating system logs are used to infer availability [31]. Such an approach also allows one to measure availability exactly, as opposed to estimating it via sampling. However, this extra knowledge is not necessarily useful for availability prediction, since very brief downtime is generally "noise" and should be ignored. For example, most downtime due to software upgrades or rejuvenation rebooting should be aperiodic (making it difficult to predict) and fairly brief (so that it has little impact on a host's overall availability profile). Thus, such events should be omitted from the history that is used to make predictions, since they will only obscure essential availability trends. If

such events are substantial contributors to system downtime, then the sampling interval can be decreased. The sampling interval should also be selected with an eye towards typical session times. For example, session times are shorter in Overnet than in PlanetLab, so Overnet nodes should be sampled more than PlanetLab ones.

5 Applications of Availability Prediction

For our first application of availability prediction, we describe a distributed hash table which preferentially stores objects on highly available nodes; the modified DHT transmits fewer objects for regeneration and has greater data availability. We then show how availability predictors can improve routing performance in delay-tolerant networks. Finally, we integrate availability prediction with models of computer virus propagation and show how we can capture diurnal fluctuations in infection intensity. For each application, we use Hybrid predictors having the parameterizations and training times described in Section 4.1.

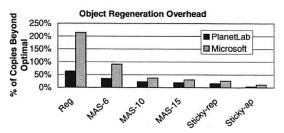
5.1 Availability-aware Replica Placement

In a distributed data store, objects are replicated for reliability and availability. When a replica site goes offline, its objects typically must be copied from another machine and regenerated at a new site. By biasing data placement towards highly available nodes, we reduce the number of objects that must be shipped for regeneration. The bandwidth savings will be substantial if objects are large or there are many objects.

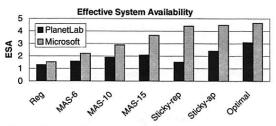
We frame our investigation of replication strategies within the context of the Chord routing infrastructure [32]. Each Chord node has a 160-bit overlay identifier, typically the hash of its IP address. The 2^{160} possible identifiers form a circular address space; the *successor* of a node is the first online machine with a larger identifier mod 2^{160} , and the *predecessor* is defined similarly. Each node tracks s immediate successors as well as several routing table peers. Through clever selection of routing table entries, nodes only need to maintain O(logN) entries to provide O(logN) route length.

In a Chord-based DHT, an object is stored on the first node whose identifier is larger than the hash of the object. If replication is desired, the k replicas are stored on the first k nodes with larger identifiers [32]. The node immediately preceding the replica sites for an object is that object's *replica manager*. Queries for that object are routed to the replica manager, who responds to the initiator with the IP addresses of the replica sites.

We investigate five replication strategies. The first two do not use availability prediction. In the regular replication method described above, objects must be regenerated whenever a replica site leaves or a node join causes the first k successors of an object id to change. In the sticky replica strategy, a newly entering node N places replicas on its first k join-time successors. N continues to use a replica site until that site leaves the overlay, at which point it is replaced with



(a) Availability-guided data placement reduces the copy overhead incurred when replica sites depart.



(b) By biasing object placement towards nodes which will be online for several consecutive hours, data availability also increases.

Figure 7: DHT simulation results

Fairness of Storage Burdens

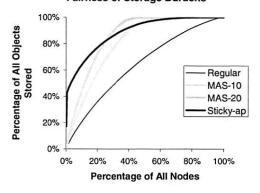


Figure 8: Storage skew in the Microsoft DHT

the first successor of N that is not already a replica site for N. The sticky replica strategy requires fewer object copies than the standard scheme since only node leaves cause replicas to be transferred. Unfortunately, the overlay identifiers for an object's replica sites are no longer a simple function of that object's id. If a replica manager goes down unexpectedly, the pointers to its replica nodes are lost, and the associated data cannot be rediscovered in an efficient way [7]. To guard against this, each node backs up its replica site pointers on its first k successors. When a node leaves the overlay, its predecessor can find the replica sites for the new objects it manages and copy the necessary data to its replica sites.

The next two replication strategies use availability prediction to guide replica placement. Each node has a Hybrid predictor that it updates every hour. After each update, the node estimates the remaining number of hours that it will remain online, making iterative availability predictions for

1 hour into the future up to some maximum lookahead period. During each iteration, the estimate is incremented by 1 if the Hybrid predictor outputs "online" and the observed accuracy of predictions for that lookahead period surpasses a minimum threshold; if either condition is false, iteration terminates. Nodes periodically exchange their estimated availabilities with the other peers in their routing tables. These values are piggybacked atop standard routing stabilization messages [32]. In the simulation results given later, the maximum lookahead period was 15 hours and the minimum confidence level was 90%.

In the *most-available successors* replication strategy, abbreviated MAS-j, a replica manager places objects on the k most available of its first j successors, where $k \leq j \leq s$. A node's replica site is sticky as long as it remains one of the first j successors. The *sticky replicas with availability prediction* scheme, abbreviated sticky-ap, features unconstrained attachment to replica sites as in the regular sticky strategy. Additionally, when a node picks a new replica site, it picks the most available of its immediate s successors that is not already a replica site.

For comparison purposes, we also study the *optimal* placement strategy. This strategy is like sticky-ap, but the availability predictor is an oracle. When a new replica site must be picked, the optimal scheme selects the node in the first s successors that will definitely be online for the longest consecutive period.

Figure 7 shows DHT performance when availability is driven by the Microsoft or the PlanetLab trace. The results were produced using a derivative of the well-known Chord simulator [32]. Leaves and joins that happened in the same hour in an availability trace were uniformly and randomly distributed across the corresponding hour in the simulation. All simulations ran for 504 virtual hours. The simulated Microsoft DHT contained 1000 nodes and the simulated Planet-Lab one contained 321 nodes. 2% of DHT operations were writes and 98% were reads. In aggregate, the Microsoft DHT issued about 660 requests each minute according to a Poisson distribution. The PlanetLab DHT used the same per-node request rate, but due to its smaller size, it only issued about 210 aggregate requests per minute. The replication factor was 4 in both DHTs, and each node's routing cache tracked 20 immediate successors. Each replication strategy was tested five times. The ith run for each strategy used the same set of nodes, but this set was changed for each value of i. Standard deviations were less than 3% amongst the trials for a particular replication strategy.

Figure 7(a) shows that availability-guided replication strategies result in many fewer copies due to replica regeneration. The savings are largest in the Microsoft DHT, with the regular replication strategy requiring 215% more copies than optimal and the sticky-ap strategy requiring only 11% more copies than optimal. The gains are smaller in the PlanetLab system, with the regular replication strategy requiring 65%

more copies than optimal and the sticky-ap strategy requiring 3.4% copies beyond optimal. The discrepancy in savings is primarily explained by the fact that the Microsoft system had more nodes that were always online. Biasing data storage towards such nodes will lower overhead more than biasing objects towards long stretch nodes that will be online for several consecutive hours, but will still eventually go offline and require object copying. The Microsoft DHT also had workweek periodic nodes, unlike the PlanetLab one. Although work-week periodic nodes may often be offline, we can still take advantage of phase-shifted diurnal patterns to reduce object copying (see the example in Section 2).

We should distinguish between the savings derived from having sticky replica sites and the savings produced by clever choice of these sites. For example, in the Microsoft DHT, if we compare the sticky-rep strategy with sticky-ap, we see that sticky-ap required roughly 7.3 million copies, whereas the sticky replicas strategy required about 8.3 million copies. This reduction of a million copies can be understood as the savings from quickly identifying highly available nodes, as opposed to hoping that your first k successors are highly available and having to regenerate their replicas if you are wrong.

Figure 7(b) describes the effective system availability (ESA) of the two DHTs using various replication strategies. ESA expresses global object availability in units of "nines." For example, if we expect an arbitrary object to be accessible via some replica site 99% of the time, the system-wide object availability is 0.99 or 2 nines of availability; more detailed discussion of ESA is provided elsewhere [9]. As expected, ESA goes up as the DHT has more freedom to bias object storage towards highly available nodes, and the improvement is greater in the Microsoft system. For example, in the Microsoft DHT, MAS-15 more than doubles the baseline ESA, adding 2.17 nines. In the PlanetLab DHT, MAS-15 improves ESA from 1.31 to 2.08.

If nodes use a replication strategy with unconstrained stickiness, a node may place objects on peers that "move beyond" its first s successors. In these scenarios, nodes will have to devote extra heartbeat messages to ensure that these replica sites are online. If objects are relatively small, then the relative cost of these heartbeat messages may justify the use of schemes such as MAS-10 which place replicas on peers which would already be pinged.

Also note that there is a tension between reducing object copies and maintaining equitable storage burdens. This tension is depicted in Figure 8, which shows the cumulative distribution of object storage with respect to the number of nodes in the Microsoft DHT. The line y=x represents a perfectly equitable storage burden, i.e., X% of the total objects would be stored on exactly X% of the nodes. The regular replication strategy was close to this line, although it was slightly convex since real-life storage burdens will never be exactly uniform. In the standard replication scheme, 1.4% of nodes

stored less than 100 objects per online hour, and 59.7% of nodes stored between 300 and 900 objects per online hour. The distributions for the availability-guided replication strategies were much less equitable. For example, with MAS-20, 57.0% of nodes stored less than 100 objects per online hour. The storage skew was even more dramatic for the sticky-ap scenario, where 10% of nodes stored 64% of the objects.

The system designer must balance competing requirements for high ESA, low bandwidth usage, and equitable storage burdens. The threat model must also be considered. If nodes are untrusted, it is unwise to bias too many objects towards a few nodes, since this multiplies an attacker's ability to corrupt the object store. Studying these trade-offs is an important area for future work.

5.2 Delay-tolerant networks

In the traditional wired Internet, machines often have the luxury of persistent, high quality connections to their peers. In contrast, *delay-tolerant networks* (DTNs) [13] are composed of heterogeneous devices with vastly differing networking and storage capabilities. Delay-tolerant networks must deliver messages in spite of intermittent device connectivity and differences in link bandwidth and latency that may span orders of magnitude. We provide two examples of DTNs later in this section.

The simplest DTN routing algorithm forwards each message along the first available link providing forward progress. Given the heterogeneous link qualities and intermittent connectivities that characterize a DTN, we would expect such a naive routing scheme to perform far worse than optimal. Jain et al describe how to use resource oracles to decrease message delivery times [16]. For example, they define a contact oracle which has perfect knowledge of link characteristics. Given two devices and an arbitrary point in the future, the contact oracle can output whether a link will exist between the two devices. Using our availability prediction techniques, we can approximate such an oracle and plug it into the routing algorithm described in [16].

We evaluate our contact oracle by simulating the behavior of two DTNs. The first one is reminiscent of an example provided by Jain et al. Imagine that a remote village without wired Internet access wishes to fetch web pages. Further suppose that the village is willing to tolerate asynchronous delivery latencies on the order of a day; such latencies are quite reasonable if the web pages are required to teach a class whose syllabus is known in advance. The remote village has a much larger sister city with a wired Internet connection, but this city is several hours away by ground transportation. Luckily, the buses that travel between the two cities can act as data mules, with outbound vehicles from the village carrying web requests and inbound vehicles carrying web data to be downloaded in the city. We assume that there are three round trips between the village and the city each day. The time required for each one way trip is chosen uniformly from

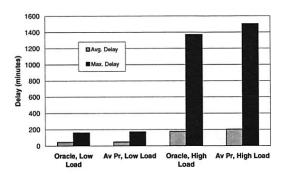
the range [100 minutes, 140 minutes]. The first bus leaves the village at 8 AM, and the last bus is expected to return to the village at 8 PM. Each bus has a data capacity of 128 MB (equivalent to a small USB memory card), and each bus stays in network contact at the village or the city for five minutes. We assume that the bandwidth of the USB device is 1 Mbps.

The village has two additional means of communication. First, the village and the city are periodically connected by a satellite link. The satellite is close enough to both locations to form a direct link every six hours, and the link persists for ten minutes. The satellite bandwidth is 10 kilobits per second and the latency is 3 seconds. Second, the village has access to a slow dial-up modem which, for reasons of expense, is only accessible from 11 PM to 6 AM. Due to an unreliable telephone infrastructure, this link is offline for 10% of its ostensibly available period, with the unexpected disconnections scattered uniformly between 11 PM and 6 AM.

Figure 9 shows simulation results for the DTN described above. Web requests were 1KB on average and web responses were 10KB on average, as suggested by empirical studies of web traffic [28]. Predictors were trained on two weeks of synthetic availability data with a sampling granularity of 20 minutes. Messages were then generated at randomly chosen times for 5 simulated days; simulation termination occurred once all messages had been delivered. Routing was reactive, i.e., when a message arrived at a node to be forwarded, the node used the most recently observed availability data to calculate the route for that message.

In the low load scenario of Figure 9, the village and the city exchanged 200 messages a day (i.e., 200 web requests were sent to the city and 200 web fetches were sent to the village). Using our availability predictors led to average message latencies that were only 6.7% worse than those incurred by optimal oracles. The worse-case delay was only 7.5% worse. In the high load scenario, the village and the city exchanged 1000 messages a day. Uptime mispredictions resulted in greater penalties in this scenario, since a single poor prediction could result in many messages being routed through an erratically online node. However, average message delays in our predictor system were still within 16% of those in a system with infallible contact oracles. Worse case delays were within 9.6% of optimal.

Our second evaluation DTN represents a collaborative sensor node system. We suppose that each user in the system possesses a laptop, a desktop PC, and a set of trusted laptops and desktops belonging to friends. When a user moves to a new location and begins to work on her laptop, the laptop may "notice" something interesting about the surrounding environment, e.g., the availability of a new wireless access point. The user wants to share this information in an effort-free (and secure) method with her friends. Thus, her laptop acts as a store-and-forward node for the interesting piece of information. If the laptop is online when a friend's device is online, the laptop can directly transmit the information to the friendly



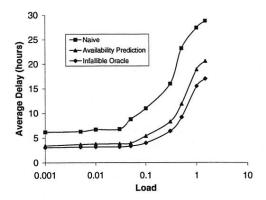
Using our availability predictors as DTN contact predictors results in message delays that are close to those generated by an infallible contact oracle. These results represent the outcomes of twenty simulation runs.

Figure 9: Message delays in the village DTN.

device. Otherwise, the laptop tries to forward the data across a path of trusted machines.

In our simulation of this DTN, each PC's uptime was driven by a trace from the Microsoft corporate network. Each laptop's availability was driven by a trace from Kim et al's wireless availability study [18]. We filtered out laptops and PCs which were not online at least once during the first 20% and the last 20% of their respective trace period. Laptop availability was sampled every 30 minutes and PC availability was sampled every hour. Each user sent messages to a trusted collection of ten laptops and ten PCs. Message sizes varied uniformly between 1KB and 20KB, and messages were randomly generated by each laptop using a Poisson distribution with a λ of 0.5 messages per online hour. When a laptop generated a message, it first delivered it to all trusted nodes which were also online. If the laptop predicted that an offline friend would come online before it left the network, it would wait to deliver the data directly. Otherwise, it would instruct one or more of its online buddies to forward the data to the remaining friendly devices. If, for a particular message, several destination devices shared a common next hop from the current machine, only one copy of the data was forwarded to the next hop. Laptops communicated with desktops using 11 Mbps wireless links, and desktops communicated with each other using 100 Mbps Ethernet connections. We assumed that a path (i.e., link) existed between two machines if they were online at the same time, and all routing was reactive.

Figure 10 shows simulation results for the collaborative sensor DTN. Each data point represents the average of 20 trials, and during each trial, the DTN was comprised of a random subset of 300 laptops and 300 PCs from the respective availability traces. Each trial ran for 504 simulated hours, and predictors were pre-trained on 336 hours of data. We used Jain *et al*'s definition of load [16], such that the load over the duration of a simulation was the sum of the traffic demand divided by the sum of the available transmission bandwidth



In low to medium load situations, our availability prediction scheme is within 15% of optimal. The performance gap widens for loads greater than 0.6, with our scheme 40% worse than optimal under a load of 1.5. However, we still perform much better than a naive routing scheme which simply waits for the sender and the receiver to come online at the same time

Figure 10: Message delays in a collaborative sensor DTN.

during this time. Note that some of this bandwidth would lie idle if a node had nothing to transmit at a particular moment.

As in the village DTN, when loads became high, the delay differences between the availability prediction system and the infallible oracle system grew. This difference was at worst 40% for a load of 1.5, but was closer to 10-15% for more reasonable loads. Additionally, our availability prediction system always had better performance than a naive scheme in which nodes never forwarded data using intermediate nodes and always waited for the destination machine to come online. Thus, we believe that our availability predictors can provide a meaningful improvement in DTN performance.

DTNs are a fairly new idea, so there is no consensus on the best way to maintain distributed DTN routing state. One could imagine that hosts with the necessary computational resources engage in a BGP-like protocol [12] to exchange link state. Each routing table entry would contain the next hop to a particular destination and the predicted availability profiles of nodes along the route. Each device would track its own availability history, but computationally weak nodes would push the tasks of uptime prediction and route selection to more powerful ones.

5.3 Virus Modeling

Traditional analytic models of computer virus propagation [17, 25, 34] assume that machines are always online. This assumption is often incorrect—the non-trivial churn rates found in real distributed systems result in network topologies with rich time-sensitive dynamics. At any given moment, some infected machines are offline (and thus effectively non-contagious), and some susceptible machines are offline (and therefore temporarily protected from infection). In such a fluid topology, the infection rate is no longer a

simple function of the virulence of the malicious code and the time it takes to discover an infected node and install the relevant software patch. Now, we must incorporate a timevarying availability function which describes node churn. Such time-dependent availability diminishes the aggressiveness of viral propagation, since infected hosts will be unable to spread the virus when they are offline. However, we cannot determine that a diseased node is sick until it comes online. Thus, the availability dynamic also impedes the discovery of infected hosts and subsequent application of the "cure." The net result is a quantitative and qualitative impact on the infection dynamic, an impact which has been observed in the real world. For example, an empirical study of the Code-Red worm discovered strong diurnal patterns in viral behavior, with the number of active diseased hosts spiking at the start of the workday and ebbing as some people applied patches and many people turned off their workplace computers and left for their homes [23].

As a first step towards more expressive viral models, we have derived an availability-aware version of the Kephart-White framework [17]. The classic Kephart-White model uses a differential equation to describe computer virus propagation. It assumes a susceptible-infected-susceptible (SIS) environment—a machine enters the system in a healthy state, and it can catch and subsequently be cured of the infection an infinite number of times 2 . The Kephart-White model assumes a homogeneous network topology in which all nodes have similar levels of connectivity or "out-degree." In such a network, the fraction f of infected nodes is given by:

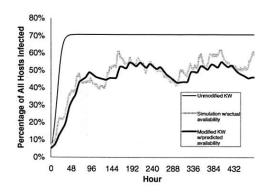
$$\frac{df}{dt} = \beta \langle k \rangle f(1 - f) - \delta f \tag{1}$$

where t is time, β is the viral birth rate along every edge from an infected node, δ is the cure rate at each infected node, and $\langle k \rangle$ is the average connectivity of a node. β, δ , and $\langle k \rangle$ are assumed to be constant.

To represent the notion of machine availability, we define a time-varying activity percentage, denoted a. Just like f, a assumes values in the range [0.0, 1.0]. At time t, we let a(t) represent the fraction of all machines in the distributed system which are currently online. This results in the following differential equation:

$$\frac{df}{dt} = \beta \langle k \rangle (fa)[(1-f)a] - \delta fa. \tag{2}$$

With respect to the standard Kephart-White equation, we replaced the infected fraction f with fa and the susceptible fraction (1-f) with (1-f)a. These new quantities represent the fact that nodes must be active to transmit or receive the virus.



Accuracy of viral models for the Microsoft corporate network (β =0.008, δ =0.07, $\langle k \rangle$ =30). The forecasts of our new model are much closer to the results of discrete time simulation driven by real availability data. For the availability-aware results, the two humps after hour 144 represent peaks in infection activity during two Monday to Friday work weeks. The lookahead period for predicted availability was 24 hours.

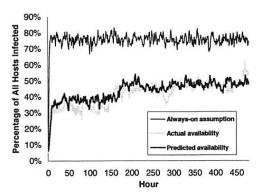
Figure 11: Epidemics in homogeneous topologies

Figure 11 shows that fluctuating node availabilities have a marked impact upon infection dynamics. Given β =0.008, δ =0.07, and $\langle k \rangle$ =30, the standard Kephart-White model predicts a steady state infection fraction of 70%. However, if we remove the erroneous assumption of constant network connectivity and use real machine availabilities, viral propagation changes in two ways. First, there are cyclical fluctuations in the number of infected nodes corresponding to the diurnal work-week patterns in the underlying availability trace. Second, the average steady state infected percentage is depressed, relative to an environment in which nodes are always on.

What explains these new phenomena? Remember that a node can be cured or infected only if it is online. Given that a node is online, the probability of being cured is proportional to the constant δ , whereas the probability of being infected is proportional to the constant β and $\langle k \rangle fa$, the number of infected neighbors that are currently online. The likelihood of being cured is unrelated to the availability of its neighbors. However, its chances of being infected will diminish if its neighbors ever go offline. Thus, the unavailability of machines effectively strengthens the cure "force." This strengthening is dependent on the rate at which machines enter and leave the network. Since this rate has diurnal fluctuations, a diurnal infection dynamic emerges.

The Kephart-White model assumes that each node has the same number of neighbors. Such assumptions of connectivity homogeneity are reasonable when analyzing malicious code that spreads indiscriminately, e.g., via random IP scanning. The homogeneity assumption may be unwarranted for viruses which spread via application-level vectors that are governed by social relationships. For example, email contact graphs have a power-law connectivity distribution, meaning that most people have few contacts and a small number of people have many contacts [11]. In these situations, epidemi-

²The simple SIS model has no conception of permanent immunity, so it only crudely models the deployment of remedies like software patches. We use the SIS model here for pedagogical clarity, but it is straightforward to incorporate availability-awareness into more realistic models such as susceptible-infected-removed.



Viral simulation results for a 321 node power-law topology and the PlanetLab availability trace (β =0.2, δ =0.24). The average node connectivity was 5.89, with a minimum degree of 3 and a maximum of 56. The lookahead period for predicted availability was 24 hours.

Figure 12: Epidemics in power-law topologies

ological models which assume contact homogeneity will have poor predictive power.

Adding availability prediction to an analytic model for power-law epidemics is an important area of unfinished work. However, we can already use simulation-based approaches to discern the impact of fluctuating uptimes upon the viral dynamic. Figure 12 shows simulation results for a 321 node power-law network with uptimes driven by the PlanetLab trace. Although global availability in the PlanetLab system showed no diurnal periodicity, more than 40% of all nodes were offline at any given moment. This should lead to a large depression in the infection level that would have resulted if machines were always online. Simulations using predicted availability captured this phenomenon well and were quantitatively similar to simulations that used actual (oracle) availability. This accuracy was achieved despite the fact that, as Figure 3(b) shows, only 17% of PlanetLab nodes are 95%predictable for a 24 hour lookahead. These results imply that availability prediction can still provide useful benefits in environments that are relatively unpredictable in the mediumto-long term.

When availability data is used to assist antiviral efforts, the dissemination mechanism for this data will depend on the configuration of the antivirus system. For example, enterprise-level antiviral systems often use a small number of servers to receive new virus definitions. These centralized servers push new definitions to clients whenever they please, and they can force clients to scan for malware. Since endusers cannot stop these forced scans, the centralized servers have complete control over enterprise-wide antiviral policy. In such a scenario, the servers can also act as the global repository for availability data. They may ping clients directly, aggregate client-reported availability data, and/or infer availability through inspection of DHCP requests, ARP traffic, etc. Using this data to predict future availability, the servers

can then prioritize patch distribution. For example, patches should preferentially be pushed to clients which are always on, since these are the machines which, if infected, will have the most opportunities to infect other hosts. If the system spans time zones, then one might want to create a "time zone firewall" by preferentially patching work-week periodic hosts in time zones where the work day is about to begin.

6 Conclusion

Loosely-coupled distributed systems are comprised of nodes that can join and leave the collective at any time. Previous models of peer-to-peer availability [2, 5, 10] provide conservative estimates of uptime, but these models cannot predict changes in availability over time. To achieve true insights into the behavior of peer-to-peer systems, availability must be a first class concern.

In this paper, we introduce new techniques for availability prediction. Our predictors track fine-grained, per-node uptime state to estimate future availability, leveraging the most accurate estimation mechanism for each situation. To quantitatively characterize differences in availability between multiple distributed systems, we use techniques from signal analysis and information theory to create uptime taxonomies.

We describe three useful applications of availability prediction. By biasing replica storage towards highly available nodes, we can reduce network bandwidth consumption and increase data availability. Using our uptime predictors as contact oracles, we can reduce message latencies in delay-tolerant networks. Finally, by incorporating availability into epidemiological models, we can capture empirically observed infection dynamics.

Acknowledgments

We thank the NSDI reviewers, and especially Doug Tygar, our shepherd, for their many helpful comments and suggestions. This work was supported in part by the National Science Foundation under Grant No. CNS-0509089. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

- A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of NSDI*, San Francisco, CA, March 2004.
- [2] R. Bhagwan, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer systems. Technical Report CS2002-0726, UCSD, November 2002.
- [3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd IPTPS*, Berkeley, CA, February 2003.
- [4] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: system support for automated availability management. In *Proceedings of NSDI*, March 2004.

- [5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *Proceedings of the 9th HotOS*, May 2003.
- [6] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of ACM SIGMETRICS*, Santa Clara, CA, June 2000.
- [7] K. Chen, L. Naamani, and K.Yehia. miChord: decoupling object lookup from placement in DHT-Based Overlays. http://www.loai-naamani.com/Academics/miChord.htm.
- [8] J. Douceur. Is remote host availability governed by a universal law? SIGMETRICS Performance Evaluation Review, 31(3):25–29, 2003.
- [9] J. Douceur and R. Wattenhofer. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proceedings of the 9th MASCOTS*, pages 311–319, Cincinnati, Ohio, August 2001.
- [10] J. Douceur and R. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th IEEE SRDS*, pages 4–13, New Orleans, LA, October 2001.
- [11] P. Drineas, M. Krishnamoorthy, M. Sofka, and B. Yener. Studying e-mail graphs for intelligence monitoring and analysis in the absence of semantic information. In *Proceedings of the Symposium on Intelligence and Security Informatics*, Tuscon, AZ, June 2004.
- [12] B. Duvall. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.
- [13] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of ACM SIGCOMM*, pages 27–34, August 2003.
- [14] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *IEEE Parallel and Distributed Com*puting Systems, pages 165–170, September 1995.
- [15] S. Haykin. Adaptive filter theory. Prentice Hall, Englewood Cliffs, NJ, 3rd edition, 1996.
- [16] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *Proceedings of ACM SIGCOMM*, pages 145–158, September 2004.
- [17] J. Kephart and S. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Computer Symposium on Research in Security and Privacy*, pages 343–359, May 1991.
- [18] M. Kim and D. Kotz. Modeling users' mobility among WiFi access points. In *Proceedings of the International Workshop* on Wireless Traffic Measurements and Modeling, pages 19–24, Seattle, WA, June 2005.
- [19] T. Kroeger and D. Long. The case for efficient file access pattern modeling. In *Proceedings of the 7th HotOS*, Rio Rico, AZ, March 1999.
- [20] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *Proceedings of the 14th IEEE SRDS*, 1995.

- [21] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proceedings of the* 2nd IPTPS, Berkeley, CA, February 2003.
- [22] S. McFarling. Combining branch predictors. Technical Note TN-36, DEC WRL, June 1993.
- [23] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an Internet worm. In *Proceedings* of the Second Internet Measurement Workshop, pages 273– 284, November 2002.
- [24] D. Nurmi, J. Brevik, and R. Wolski. Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments. In *Proceedings of EUROPAR*, 2005.
- [25] R. Pastor-Satorras and A. Vespignani. Epidemic Spreading in Scale-Free Networks. *Physics Review Letters*, 86(14):3200– 3203, April 2001.
- [26] S. Pincus. Approximate entropy as a measure of system complexity. In *Proceedings of the National Academy of Science*, pages 2297–2301, USA, March 1991.
- [27] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in C*. Cambridge University Press, New York, NY, 2nd edition, 1992.
- [28] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An analysis of internet content delivery systems. In *Proceedings* of OSDI, pages 315–327, December 2002.
- [29] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Mul*timedia Computing and Networking Conference, pages 18–25, January 2002.
- [30] T. Schwarz, Q. Xin, and E. Miller. Availability in global peer-to-peer storage systems. In *Proceedings of the 2nd IPTPS*, Lausanne, Switzerland, July 2004.
- [31] C. Simache and M. Kaaniche. Measurement-based Availability Analysis of Unix Systems in a Distributed Environment. In Proceedings of the International Symposium on Software Reliability Engineering, pages 346–355, Hong Kong, China, November 2001.
- [32] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalabale peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, August 2001.
- [33] J. Stribling. All-pairs PlanetLab Ping Data. http://www.pdos.lcs.mit.edu/~strib/pl_app/.
- [34] Y. Wang, D. Chakrabarti, C. Wang, and C. Faloutsos. Epidemic Spreading in Real Networks: An Eigenvalue Viewpoint. In Proceedings of the Symposium on Reliable Distributed Computing, pages 25–34, Florence, Italy, October 2003.
- [35] Rich Wolski. Experiences with predicting resource performance on-line in computational grid settings. SIGMETRICS Performance Evaluation Review, 30(4):41–49, 2003.

To Infinity and Beyond: Time-Warped Network Emulation

Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker University of California, San Diego

{dgupta,kyocum,mmcnett,snoeren,vahdat,voelker}@cs.ucsd.edu

Abstract

The goal of this work is to subject unmodified applications running on commodity operating systems and stock hardware to network speeds orders of magnitude faster than available at any given point in time. This paper describes our approach to *time dilation*, a technique to uniformly and accurately slow the passage of time from the perspective of an operating system by a specified factor. As a side effect, physical devices—including the network—appear relatively faster to both applications and operating systems. Both qualitative and statistical evaluations indicate our prototype implementation is accurate across several orders of magnitude. We demonstrate time dilation's utility by conducting high-bandwidth head-to-head TCP stack comparisons and application evaluation.

1 Introduction

This work explores the viability and benefits of *time dilation*—providing the illusion to an operating system and its applications that time is passing at a rate different from physical time. For example, we may wish to convince a system that, for every 10 seconds of wall clock time, only one second of time passes in the operating system's dilated time frame. Time dilation does not, however, change the arrival rate of physical events such as those from I/O devices like a network interface or disk controller. Hence, from the operating system's perspective, physical resources appear 10 times faster: in particular, data arriving from a network interface at a physical rate of 1 Gbps appears to the OS to be arriving at 10 Gbps.

We refer to the ratio between the rate at which time passes in the physical world to the operating system's perception of time as the *time dilation factor*, or TDF; a TDF greater than one indicates the external world appears faster than it really is. Figure 1 illustrates the

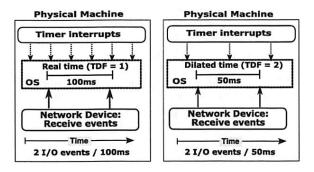


Figure 1: This figure compares a system operating in real time (left) and a system running with a TDF of 2 (right). Note that time dilation does not affect the timing of external events, such as network packet arrival.

difference between an undilated (TDF of 1) operating system on the left and a dilated OS with TDF of 2 on the right. The same period of physical time passes for both machines. Each OS receives external events such as timer (on top) and device (on bottom) interrupts. Timer interrupts update the operating systems' notion of time; in this example, time dilation halves the frequency of delivered timer interrupts.

Critically, physical devices such as the network continue to deliver events at the same rate to both OSes. The dilated OS, therefore, perceives twice the network I/O rate because it experiences only half the delay between I/O events. In particular, the "undilated" OS observes a delay of 100 ms between packet arrivals, while the dilated OS observes only 50 ms between packet arrivals. From the dilated frame of reference, time dilation scales the observed I/O rate by the TDF (in this case two).

Interestingly, time dilation scales the perceived available *processing* power as well. A system will experience TDF times as many cycles per perceived second from the processor. Such *CPU scaling* is particularly relevant to CPU-bound network processing because the number of cycles available to each arriving byte remains constant. For instance, a machine with TDF of 10 sees a 10 times

faster network, but would also experience a tenfold increase in CPU cycles per second.

By employing large TDF values, time dilation enables external stimuli to appear to take place at higher rates than would be physically possible, presenting a number of interesting applications. Consider the following scenarios:

- Emerging I/O technologies. Imagine a complex cluster-based service interconnected by 100-Mbps and 1-Gbps Ethernet switches. The system developers suspect overall service performance is limited by network performance. However, upgrading to 10-GigE switches and interfaces involves substantial expense and overhead. The developers desire a low-cost mechanism for determining the potential benefits of higher performance network interconnects before committing to the upgrade.
- Scalable network emulation. Today large ISPs cannot evaluate the effects of modifications to their topology or traffic patterns outside of complex and high-level simulations. While they would like to evaluate internal network behavior driven by realistic traffic traces, this often requires accurate emulation of terabits per second of bisection bandwidth.
- High bandwidth-delay networking. We have recently seen the emergence of computational grids [4, 12] inter-connected by high-speed and high-latency wide-area interconnects. For instance, 10-Gbps links with 100-200 ms round-trip times are currently feasible. Unfortunately, existing transport protocols, such as TCP, deliver limited throughput to one or more flows sharing such a link. A number of research efforts have proposed novel protocols for high bandwidth-delay product settings [11, 14, 16-18, 28, 30]. However, evaluation of the benefits of such efforts is typically relegated to simulation or to those with access to expensive wide-area links.

This work develops techniques to perform accurate time dilation for unmodified applications running on commodity operating systems and stock hardware with the goal of supporting the above scenarios. To facilitate dilating time perceived by a host, we utilize virtual machine (VM) technology. Virtual machines traditionally have been used for their isolation capabilities: applications and operating systems running inside a VM can only access physical hardware through the virtual machine monitor. This interposition provides the additional potential to independently dilate time for each hosted virtual machine. In this paper, we are particularly interested in time dilation with respect to network devices; we leave faithful scaling of other subsystems to future work.

This paper makes the following contributions:

- Time-dilated virtual machines. We show how to use virtual machines to completely encapsulate a host running a commodity operating system in an arbitrarily dilated time frame. We allow processing power and I/O performance to be scaled independently (e.g., to hold processing power constant while scaling I/O performance by a factor of 10, or vice versa).
- Accurate network dilation. We perform a detailed comparison of TCP's complex end-to-end protocol behavior—in isolation, under loss, and with competing flows—under dilated and real time frames. We find that both the micro and macro behavior of the system are indistinguishable under dilation. To demonstrate our ability to predict the performance of future hardware scenarios, we show that the time-dilated performance of an appropriately dilated sixyear old machine with 100-Mbps Ethernet is indistinguishable from a modern machine with Gigabit Ethernet.
- End-to-end experimentation. Finally we demonstrate the utility of time dilation by experimenting with a content delivery overlay service. In particular, we explore the impact of high-bandwidth network topologies on the performance of BitTorrent [9], emulating multi-gigabit bisection bandwidths using a traffic shaper whose physical capacity is limited to 1Gbps.

The remainder of the paper is organized as follows: We present our prototype implementation in Section 2. We evaluate the accuracy of time dilation with comprehensive micro-benchmarks in Section 3 and present application results in Section 4. Section 5 presents related work before concluding in Section 6.

2 Design and implementation

Before describing the details of our implementation, we first define some key terminology. A *Virtual Machine (VM)* or *domain* is a software layer that exports the interface of a target physical machine. Multiple VM's may be multiplexed on a single physical machine. A *Guest OS* is the Operating System that runs within a VM. Finally, a *Virtual Machine Monitor (VMM)* or *hypervisor* is the hardware/software layer responsible for multiplexing several VMs on the same physical machine.

Critical to time dilation is a VMM's ability to modify the perception of time within a guest OS. Fortunately, VMMs must already perform this functionality, for example, because a guest OS may develop a backlog of "lost ticks" if it is not scheduled on the physical

processor when it is due to receive a timer interrupt. VMMs typically periodically synchronize the guest OS time with the physical machine's clock. One challenge is that operating systems often use multiple time sources for better precision. For example, Linux uses up to five different time sources [19]. Exposing so many different mechanisms for time keeping in virtual machines becomes challenging (see [27] for a discussion).

To be useful, time dilation must be pervasive and transparent. Pervasiveness implies that the system is completely isolated from the passage of physical time. Similarly, transparency implies that network protocols and applications require no modification to be used in a dilated time frame. To address these requirements, we implemented a time dilation prototype using the Xen VMM [7] (Our implementation is based on Xen 2.0.7). We chose Xen for the following reasons: (1) it is easier to modify behavior of timer interrupts in software than in hardware; (2) we can observe time dilation in isolated, controlled environments; (3) Xen allows us to provide each virtual machine with an independent time frame; (4) Xen source code is publicly available; and iv) the VMM CPU scheduler provides a facility for scaling CPU. Though our implementation is Xen-specific, we believe the concepts can apply to other virtual machine environments.

Alternative implementation targets for time dilation include directly modifying the operating system, simulation packages, and emulation environments. As discussed below, our modifications to Xen are compact and portable, giving us confidence that our techniques will be applicable to any operating system that Xen supports. In some sense, time dilation is free in many simulation packages: extrapolating to future scenarios is as simple as setting appropriate bandwidth values on particular links. However, we explicitly target running unmodified applications and operating systems for necessary realism. Finally, while network emulation does allow experimentation with a range of network conditions, it is necessarily limited by the performance of currently available hardware. For this reason, time dilation is a valuable complement to network emulation, allowing an experimenter to easily extrapolate evaluations to future, faster environments.

We now give a brief overview of time keeping in Xen, describe our modifications to it, and discus the applicability of time dilations to other virtualization platforms.

2.1 Time flow in Xen

The Xen VMM exposes two notions of time to VMs. *Real time* is the number of nanoseconds since boot. *Wall clock time* is the traditional Unix time-since-epoch (midnight, January 1, 1970 UTC). Xen also delivers periodic

timer interrupts to the VM to support the time keeping mechanisms within the guest OS.

While Xen allows the guest OS to maintain and update its own notion of time via an external time source (such as NTP), the guest OS often relies solely on Xen to maintain accurate time. Real and wall clock time pass between the Xen hypervisor and the guest operating system via a shared data structure. There is one data structure per VM written by the VMM and read by the guest OS.

The guest operating system updates its local time values on demand using the shared data structure — for instance, when servicing timer interrupts or calling getttimeofday. However, the VMM updates the shared data structure only at certain discrete events, and thus it may not always contain the current value. In particular, the VMM updates the shared data structure when it delivers a timer interrupt to the VM or schedules the VM to run on an available CPU.

Xen uses *paravirtualization* to achieve scalable performance with virtual machines without sacrificing functionality or isolation. With paravirtualization, Xen does not provide a perfect virtualization layer. Instead, it exposes some features of the underlying physical hardware to gain significant performance benefits. For instance, on the x86 architecture, Xen allows guest OSes (for our tests, we use XenoLinux as our guest OS) to read the Time-Stamp Count (TSC) register directly from hardware (via the RDTSC instruction).

The TSC register stores the number of clock cycles since boot and is incremented on every CPU cycle. The Guest OS reads the TSC to maintain accurate time between timer interrupts. By contrast, kernel variables such as Linux jiffies or BSD ticks only advance on timer interrupts. In this case, we modify the guest OS to prevent them from reading the true value of the TSC, as described in the next section.

2.2 Dilating time in Xen

We now outline our modifications to the Xen hypervisor and the XenoLinux kernel to support time dilation. We focus on slowing down the passage of time so that the external world appears faster. It is also possible to speed the passage of time from the OS's perspective, thereby slowing processes in the external world. In general, however, speeding the passage of time is less useful for our target scenarios and we do not explore it in this paper.

Our modifications to Xen are small: in all, we added/modified approximately 500 lines of C and Python code. More than 50% of our changes are to non-critical tools and utilities; the core changes to Xen and Xeno-Linux are less than 200 lines of code. Our modifications are less than 0.5% of the base code size of each component.

Variable	Original	Dilated	
Real time	stime_irq	stime_irq/tdf	
Wall clock	wc_sec,	wc_sec/tdf,	
	wc_usec	wc_usec/tdf	
Timer interrupts	HZ/sec	(HZ/tdf)/sec	

Table 1: Basic Dilation Summary

Modifications to the Xen hypervisor. Our modified VMM maintains a TDF variable for each hosted VM. For our applications, we are concerned with the relative passage of time rather than the absolute value of real time; in particular, we allow—indeed, require—that the host's view of wall clock time diverge from reality. Thus the TDF divides both real and wall clock time.

We modify two aspects of the Xen hypervisor. First we extend the shared data structure to include the TDF field. Our modified Xen tools, such as xm, allow specifying a positive, integral value for the TDF on VM creation. When the hypervisor updates the shared data structure, it uses this TDF value to modify real and wall clock time. In this way, real time is never exposed to the guest OS through the shared data structure.

Dilation also impacts the frequency of timer interrupts delivered to the VM. The VMM controls the frequency of timer interrupts delivered to an undilated VM (timer interrupts/second); in most OS's a HZ variable, set at compile time, defines the number of timer interrupts delivered per second. For transparency, we need to maintain the invariant that HZ accurately reflects the number of timer interrupts delivered to the VM during a second of dilated time. Without adjusting timer interrupt frequency, the VMM will deliver TDF-times too many interrupts. For example, the VMM will deliver HZ interrupts in one physical time second, which will look to the dilated VMM as HZ/(second/TDF) = TDF*HZ. Instead, we reduce the number of interrupts a VM receives by a factor of TDF (as illustrated earlier in Figure 1). Table 1 summarizes the discussion so far.

Finally, Xen runs with a default HZ value of 100, and configures guests with the same value. However, HZ=100 gives only a 10-ms precision on system timer events. In contrast, current 2.6 series of Linux kernels uses a HZ value of 1000 by default—the CPU overhead is not significant, but the accuracy gains are tenfold. This increase in accuracy is desirable for time dilation because it enables guests to measure time accurately even in the dilated time frame. Thus, we increase the HZ value to 1000 in both Xen and the guest OS.

Modifications to XenoLinux. One goal of our implementation was to minimize required modifications to the guest OS. Because the VMM appropriately updates the shared data structure, one primary aspect of OS time-keeping is already addressed. We further modify the

guest OS to read an appropriately scaled version of the hardware Time Stamp Counter (TSC). XenoLinux now reads the TDF from the shared data structure and adjusts the TSC value in the function get_offset_tsc.

The Xen hypervisor also provides guest OS programmable alarm timers. Our last modification to the guest OS adjusts the programmable timer events. Because guests specify timeout values in their dilated time frames, we must scale the timeout value back to physical time. Otherwise they may set the timer for the wrong (possibly past) time.

2.3 Time dilation on other platforms

Architectures: Our implementation should work on all platforms supported by Xen. One remark regarding transparency of time dilation to user applications on the x86 platform is in order: recall that we intercept calls to read the TSC within the guest kernel. However, since the RDTSC instruction is not a privileged x86 instruction, guest user applications might still issue assembly instructions to read the hardware TSC register. It is possible to work around this by watching the instruction stream emanating from a VM and trapping to the VMM on a RDTSC instruction, and then returning the appropriate value to the VM. However, this approach would go against Xen's paravirtualization philosophy.

Fortunately, the current generation of x86-compatible hardware (such as the AMD Pacifica [6] and Intel VT [13]) provides native virtualization support, making it possible to make time dilation completely transparent to applications. For instance, both VT and Pacifica have hardware support for trapping the RDTSC instruction.

VMMs: The only fundamental requirement from a VMM for supporting time dilation is that it have mechanisms to update/modify time perceived by a VM. As mentioned earlier, due to the difficulties in maintaining time within a VM, all VMMs already have similar mechanisms so that they can periodically bring the guest OS time in sync with real time. For instance, VMWare has explicit support for keeping VMs in a "fictitious time frame" that is at a constant offset from real time [27]. Thus, it should be straightforward to implement time dilation for other VMMs.

Operating systems: Our current implementation provides dilation support for XenoLinux. Our experience so far and a preliminary inspection of the code for other guest OSes indicate that all of the guest OSes that Xen supports can be easily modified to support time dilation. It is important to note that modifying the guest OSes is not a fundamental requirement. Using binary rewriting,

it would be possible to use unmodified guest OS executables. We expect that with better hardware and operating system support for virtualization, unmodified guests would be able to run under dilation.

2.4 Limitations

This section discusses some of the limitations of time dilation. One obvious limitation is time itself: a 10-second experiment would run for a 100 seconds for a dilation factor of 10. Real-life experiments running for hours are not uncommon, so the time required to run experiments at high TDFs is substantial. Below we discuss other, more subtle limitations.

2.4.1 Other devices and nonlinear scaling

Time dilation uniformly scales the perceived performance of all system devices, including network bandwidth, perceived CPU processing power, and disk and memory I/O. Unfortunately, scaling all aspects of the physical world is unlikely to be useful: a researcher may wish to focus on scaling certain features (e.g., network bandwidth) while leaving others constant. Consequently, certain aspects of the physical world may need to be rescaled accordingly to achieve the desired effect.

Consider TCP, a protocol that depends on observed round-trip times to correctly compute retransmission timeouts. These timing measurements must be made in the dilated time frame. Because time dilation uniformly scales the passage of time, it not only increases perceived bandwidth, it also decreases perceptions of round-trip time. Thus, a network with 10-ms physical RTT would appear to have 1-ms RTT to dilated TCP. Because TCP performance is sensitive to RTT, such a configuration is likely undesirable. To address this effect, we independently scale bandwidth and RTT by using network emulation [23, 26] to deliver appropriate bandwidth and latency values. In this example, we increase link delay by a factor of 10 to emulate the jump in bandwidth-delay product one expects from the bandwidth increase.

In this paper, we show how to apply time dilation to extrapolate to future network environments, for instance with a factor of 10 or 100 additional bandwidth while accounting for variations in CPU power using the VMM scheduler. However, we do not currently account for the effects of increased I/O rates from memory and disk.

Appropriately scaling disk performance is a research challenge in its own right. Disk performance depends on such factors as head and track-switch time, SCSI-bus overhead, controller overhead, and rotational latency. A simple starting point would be to vary disk performance assuming a linear scaling model, but this could potentially violate physical properties inherent in disk drive

mechanics. Just as we introduced appropriate network delays to account for non-linear scaling of the network, accurate disk scaling would require modifying the virtual machine monitor to integrate a disk simulator modified to understand the dilated time frame. A well-validated disk simulator, such as DiskSim [8], could be used for this purpose. However, we leave dilating time for such devices to future work.

Finally, hardware and software architectures may evolve in ways that time dilation cannot support. For instance, consider a future host architecture with TCP offload [20], where TCP processing largely takes place on the network interface rather than in the protocol stack running in the operating system. Our current implementation does not dilate time for firmware on network interfaces, and may not extend to other similar architectures.

2.4.2 Timer interrupts

The guest reads time values from Xen through a shared data structure. Xen updates this structure every time it delivers a timer interrupt to the guest. This happens on the following events: (1) when a domain is scheduled; (2) when the *currently executing* domain receives a periodic timer interrupt; and (3) when a guest-programmable timer expires.

We argued earlier that, for successful dilation, the number of timer interrupts delivered to a guest should be scaled down by the TDF. Of these three cases, we can only control the second and the third. Our implementation does not change the scheduling pattern of the virtual machines for two reasons. First, we do not change the schedule pattern because scheduling parameters are usually global and system wide. Scaling these parameters on a per-domain basis would likely break the semantics of many scheduling schemes. Second, we would like to retain scheduling as an independent variable in our system, and therefore not constrain it by the choice of TDF. One might want to use round robin or borrowed virtual time [10] as the scheduling scheme, independent of the TDF. In practice, however, we find that not controlling timer scheduling does not impact our accuracy for all of the schedulers that currently ship with Xen.

2.4.3 Uniformity: Outside the dilation envelope

Time dilation cannot affect notions of time outside the dilation envelope. This is an important limitation; we should account for all packet processing or delays external to the VM. The intuition is that all stages of packet processing should be uniformly dilated. In particular, we scale the time a packet spends inside the VM (since it measures time in the dilated frame) and the time a packet spends over the network (by scaling up the time

on the wire by TDF). However, we do not scale the time a packet spends inside the Xen hypervisor and *Domain-O* (the privileged, management domain that hosts the actual device drivers), or the time it takes to process the packet at the other end of the connection.

These unscaled components may affect the OS's interpretation of round trip time. Consider the time interval between a packet and its ACK across a link of latency R scaled by S, and let δ denote the portion of this time that is unscaled. In a perfect world where everything is dilated uniformly, a dilated host would measure the interval to be simply $T_{perfect} = R + \delta$. A regular, undilated host measures the interval as $T_{normal} = S \times R + \delta$; a dilated host in our implementation would observe the same scaled by S, so $T_{dilated} = T_{normal}/S = (S \times R + \delta)/S$.

We are interested in the error relative to perfect dilation:

$$\epsilon = \frac{(T_{perfect} - T_{dilated})}{T_{perfect}}$$
$$= \left(1 - \frac{1}{S}\right) \left(\frac{\delta}{R + \delta}\right)$$

Note that ϵ approaches $\delta/(R+\delta)$ when S is large. In the common case this is of little consequence. For the regime of network configurations we are most interested in (high bandwidth-delay product networks), the value of R is typically orders of magnitude higher than the value of δ . As our results in Section 3 show, dilation remains accurate over a wide range of round trip times, bandwidths, and time-dilation factors that we consider.

3 Micro-benchmarks

We establish the accuracy of time dilation through a variety of micro-benchmarks. We begin by evaluating the accuracy of time dilation by comparing predictive results using dilated older hardware with actual results using undilated recent hardware. We then compare the behavior of a single TCP flow subject to various network conditions under different time dilation factors. Finally, we evaluate time dilation in more complex settings (multiple flows, multiple machines) and address the impact of CPU scaling.

3.1 Hardware validation

We start by evaluating the predictive accuracy of time dilation using multiple generations of hardware. One of the key motivations for time dilation is as a predictive tool, such as predicting the performance and behavior of protocol and application implementations on future higher-performance network hardware. To validate time dilation's predictive accuracy, we use dilation on older

Configuration	TDF	Mean (Mbps)	St.Dev. (Mbps)
2.6 GHz, 1-Gbps NIC (restricted to 500-Mbps)	1	9.39	1.91
1.13 GHz, 1-Gbps NIC (restricted to 250-Mbps)	2	9.57	1.76
500 MHz, 100-Mbps NIC	5	9.70	2.04
500 MHz scaled down to 50 MHz, 10-Mbps NIC	50	9.25	2.20

Table 2: Validating performance prediction: the mean perflow throughput and standard deviations of 50 TCP flows for different hardware configurations.

hardware to predict TCP throughput as if we were using recent hardware. We then compare the predicted performance with the actual performance when using recent hardware.

We use time dilation on four hardware configurations, listed in Table 2, such that each configuration resembles a 2.5-GHz processor with a 500-Mbps NIC, under the coarse assumption that CPU performance roughly scales with processor frequency. The base hardware configurations are 500-MHz and 1.13-GHz Pentium III machines with 10/100-Mbps and 1-Gbps network interfaces, respectively, and a 2.6-GHz Pentium IV machine with a 1-Gbps network interface. In cases where the exact base hardware was not available (a 250-Mbps NIC or a 50-MHz CPU, for instance), we scaled them appropriately using either network emulation (Dummynet [23]) or VMM scheduling (Section 3.4).

For each hardware configuration, we measured the TCP throughput of 50 flows communicating with another machine with an identical configuration. Using Dummynet, we configured the network between the hosts to have an effective RTT of 80 ms. We then calculated the mean per-flow throughput and standard deviation across all flows. Both the mean and deviation of per-flow throughput are consistent across the hardware configurations, which span over an order of magnitude of difference in hardware performance. For example, time dilation using an effective 50 MHz CPU with a 10-Mbps NIC dilated with a TDF of 50 is able to accurately predict TCP throughput of a 2.6-GHz CPU with a 1-Gbps NIC. As a result, we conclude that time dilation is an effective tool for making reasonable predictions of high-level performance on future hardware trends.

3.2 Single flow packet-level behavior

Next we illustrate that time dilation preserves the perceived packet-level timing of TCP. We use two end hosts directly connected through a Dell Powerconnect 5224 gigabit switch. Both systems are Dell PowerEdge 1750 servers with dual Intel 2.8-GHz Xeon processors, 1 GB of physical memory, and Broadcom NetXtreme

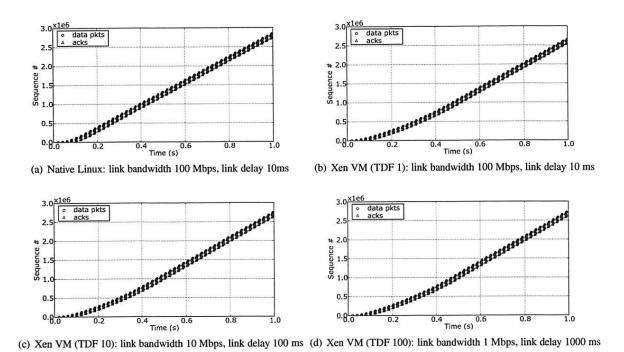


Figure 2: Packet timings for the first second of a TCP connection with no losses for native Linux and three time dilation configurations. In all cases, we configure link bandwidth and delay such that the bandwidth-delay product is constant.

BCM5704 integrated gigabit Ethernet NICs. The end hosts run Xen 2.0.7, modified to support time dilation. We use Linux 2.6.11 as the Xen guest operating system, and all experiments run inside of the Linux guest VMs. All protocols in our experiments use their default parameters unless otherwise specified. We use two identical machines running Linux 2.6.10 and Fedora Core 2 for our "unmodified Linux" results.

We control network characteristics such as bandwidth, delay, and loss between the two hosts using Dummynet. In addition to its random loss functionality, we extended Dummynet to support deterministic losses to produce repeatable and comparable loss behavior. Unless otherwise noted, all endpoints run with identical parameters (buffer sizes, TDFs, etc.).

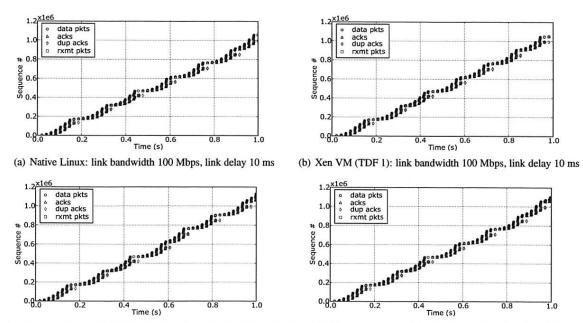
In this experiment, we first transfer data on TCP connections between two unmodified Linux hosts and use tcpdump [5] on the sending host to record packet behavior. We measure TCP behavior under both lossless and deterministic lossy conditions.

We then repeat the experiment with the sending host running with TDFs of {1, 10, 100}, spanning two orders of magnitude. When dilating time, we configure the underlying network such that a time-dilated host perceives the same network conditions as the original TCP experiment on unmodified hosts. For example, for the experiment with unmodified hosts, we set the bandwidth between the hosts to 100 Mbps and the delay to 10 ms.

To preserve the same network conditions perceived by a host dilated by a factor of 10, we reduce the bandwidth to 10 Mbps and increase the delay to 100 ms using Dummynet. Thus, if we are successful, a time dilated host will see the same packet timing behavior as the unmodified case. We include results with TDF of 1 to compare TCP behavior in an unmodified host with the behavior of our system modified to support time dilation.

We show sets of four time sequence graphs in Figures 2 and 3. Each graph shows the packet-level timing behavior of TCP on four system configurations: unmodified Linux, and Linux running in Xen with our implementation of time dilation operating under TDFs of 1, 10, and 100. The first set of graphs shows the first second of a trace of TCP without loss. Each graph shows the data and ACK packet sequences over time, and illustrates TCP slow-start and transition to steady-state behavior. Qualitatively, the TCP flows across configurations have nearly identical behavior.

Comparing Figures 2(a) and 2(b), we see that a dilated host has the same packet-level timing behavior as an unmodified host. More importantly, we see that time dilation accurately preserves packet-level timings perceived by the dilated host. Even though the configuration with a TDF of 100 has network conditions two orders of magnitude different from the base configuration, time dilation successfully preserves packet-level timings.



(c) Xen VM (TDF 10): link bandwidth 10 Mbps, link delay 100 ms (d) Xen VM (TDF 100): link bandwidth 1 Mbps, link delay 1000 ms

Figure 3: Packet timings for the first second of a TCP connection with 1% deterministic losses.

Time dilation also accurately preserves packet-level timings under lossy conditions. The second set of time sequence graphs in Figure 3 shows the first second of traces of TCP experiencing 1% loss. As with the lossless experiments, the TCP flows across configurations have nearly identical behavior even with orders of magnitude differences in network conditions.

We further evaluated the performance of a single TCP flow under a wide range of time dilation factors, network bandwidths, delays and loss rates with similar results. For brevity, we omit those results.

Figures 2 and 3 illustrate the accuracy of time dilation qualitatively. For a more quantitative analysis, we compared the distribution of the inter-arrival packet reception and transmission times for the dilated and undilated flows. Figure 4 plots the cumulative distribution function for inter-packet transmission times for a single TCP flow across 10 runs under both lossy and lossless conditions. Visually, the distributions track closely. Table 3 presents a statistical summary for these distributions, the mean and two indices of dispersion — the coefficient of variance (CoV) and the inter quartile range (IQR) [15]. An index of dispersion indicates the variability in the given data set. Both CoV and IQR are unit-less, i.e., they take the unit of measurement out of variability consideration. Therefore, the absolute values of these metrics is not of concern to us, but that they match under dilation is. Given the inherent variability in TCP, we find this similarity satisfactory. The results for inter-packet reception times are similar.

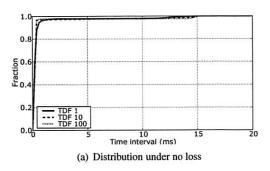
Metric	No loss			1% loss		
	TDF	TDF	TDF	TDF	TDF	TDF
	1	10	100	1	10	100
Mean (ms)	0.458	0.451	0.448	0.912	1.002	0.896
CoV	0.242	0.218	0.206	0.298	0.304	0.301
IQR	0.294	0.248	0.239	0.202	0.238	0.238

Table 3: Statistical summary of inter-packet transmission times.

3.3 Dilation with multiple flows

To demonstrate that dilation preserves TCP behavior under a variety of conditions, even for short flows, we performed another set of experiments under heterogeneous conditions. In these experiments, 60 flows shared a bottleneck link. We divided the flows into three groups of 20 flows, with each group subject to an RTT of 20 ms, 40 ms, or 60 ms. We also varied the bandwidth of the bottleneck link from 10 Mbps to 600 Mbps. We conducted the experiments for a range of flow lengths from 5 seconds to 60 seconds and verified that the results were consistent independent of flow duration.

We present data for one set of experiments where each flow lasts for 10 seconds. Figure 5 plots the mean and standard deviation across the flows within each group for TDFs of 1 (regular TCP) and 10. To visually differentiate results in each graph for different TDFs, we slightly offset their error bars on the graph although in practice they experienced the same network bandwidth conditions. For all three groups, the results from dilation



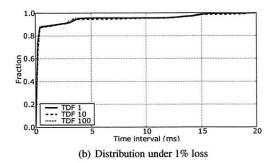


Figure 4: Comparison of inter-packet transmission times for a single TCP flow across 10 runs.

agree well with the undilated results: the throughputs for TDF of 1 match those for TDF of 10 within measured variability. Note that these results also reflect TCP's known throughput bias towards flows with smaller RTTs.

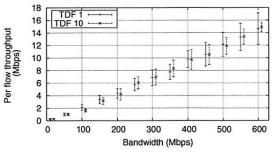
In our experiments thus far, all flows originated at a single VM and were destined to a single VM. However, when running multiple VMs (as might be the case to support, for instance, scalable network emulation experiments [26, 29]) one has to consider the impact of VMM scheduling overhead on performance. To explore the issue of VMM scheduling, we investigate the impact of spreading flows across multiple dilated virtual machines running on the same physical host. In particular, we verify that simultaneously scheduling multiple dilated VMs does not negatively impact the accuracy of dilation.

In this experiment, for a given network bandwidth we create 50 connections between two hosts with a lifetime of 1000 RTTs and measure the resulting throughput of each connection. We configure the network with an 80 ms RTT, and vary the perceived network bandwidth from 0–4 Gbps using 1-Gbps switched Ethernet. Undilated TCP has a maximum network bandwidth of 1 Gbps, but time dilation enables us to explore performance beyond the raw hardware limits (we revisit this point in Section 4.1). We repeat this experiment with the 50 flows split across 2, 5 and 10 virtual machines running on one physical machine.

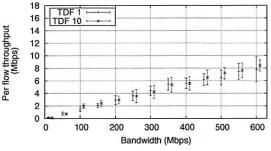
Our results indicate that VMM scheduling does not significantly impact the accuracy of dilation. Figure 6 plots the mean throughput of the flows for each of the four configurations of flows divided among virtual machines. Error bars mark the standard deviation. Once again, the mean flow throughput for the various configurations are similar.

3.4 CPU scaling

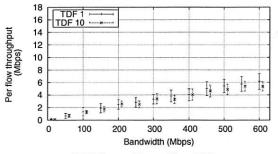
Time dilation changes the perceived VM cycle budget; a dilated virtual machine sees TDF times as many CPU cycles per second. Utilizing VMM CPU schedulers, how-



(a) 20 flows subject to 20-ms RTT



(b) 20 flows subject to 40-ms RTT



(c) 20 flows subject to 60-ms RTT

Figure 5: Per-flow throughput for 60 flows sharing a bottleneck link. Each flow lasts 10 seconds. The mean and deviation are taken across the flows within each group. To visually differentiate results in each graph for different TDFs, we slightly offset their error bars on the graph.

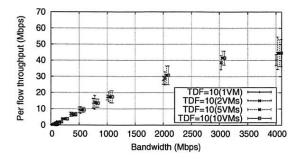


Figure 6: Mean throughput of 50 TCP flows between two hosts on a network with an 80ms RTT as a function of network bandwidth. The 50 flows are partitioned among 1–10 virtual machines.

ever, we can scale available processing power independently from the network. This flexibility allows us to evaluate the impact of future network hardware on current processor technology. In a simple model, a VM with TDF of 10 running with 10% of the CPU has the same per-packet cycle budget as an undilated VM running with 100% of the CPU. We validate this hypothesis by running an experiment similar to that described for Figure 6. This time, however, we adjust the VMM's CPU scheduling algorithm to restrict the amount of CPU allocated to each VM. We use the Borrowed Virtual Time [10] scheduler in Xen to assign appropriate weights to each domain, and a CPU intensive job in a separate domain to consume surplus CPU.

First, we find an undilated scenario that is CPU-limited by increasing link capacity. Because the undilated processor has enough power to run the network at line speed, we reduce its CPU capacity by 50%. We compare this to a VM dilated by TDF of 10 whose CPU has been scaled to 5%. The experimental setup is identical to that in Figure 6: 50 flows, 80ms RTT. For clarity, we first throttled the sender alone, leaving the CPU unconstrained at the receiver; we then repeat the experiment with the receiver alone throttled. Figures 7 and 8 show the results. We plot the per-flow throughput, and error bars mark the standard deviation.

If we successfully scale the CPU, flows across a dilated link of the same throughput will encounter identical CPU limitations. Both figures confirm the effectiveness of CPU scaling, as the 50% and 5% lines match closely. The unscaled line (100%) illustrates the performance in a CPU-rich environment. Moreover our system accurately predicts that receiver CPU utilizations are higher than the sender's, confirming that it is possible to dilate CPU and network independently by leveraging the VMMs CPU scheduling algorithm.

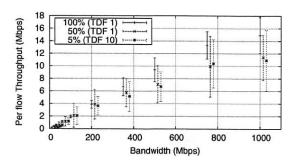


Figure 7: Per-flow throughput of 50 TCP flows between a CPU-scaled sender and unconstrained receiver. CPU utilization at the sender is restricted to the indicated percentages.

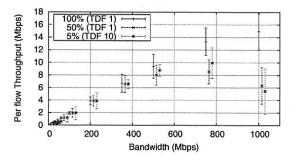


Figure 8: Per-flow throughput of 50 TCP flows between an unconstrained sender and a CPU-scaled receiver. CPU utilization at the receiver is restricted to the indicated percentages.

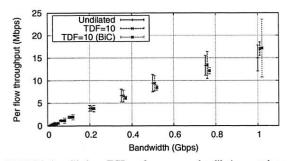
4 Applications of dilation

Having performed micro-benchmarks to validate the accuracy of time dilation, we now demonstrate the utility of time dilation for two scenarios: network protocol evaluation and high-bandwidth applications.

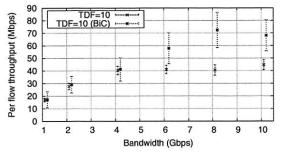
4.1 Protocol evaluation

A key application of time dilation is for evaluating the behavior and performance of protocols and their implementations on future networks. As an initial demonstration of our system's utility in this space, we show how time dilation can support evaluating optimizations to TCP for high bandwidth-delay network environments, in particular using the publicly available BiC [30] extension to the Linux TCP stack. BiC uses binary search to increase the congestion window size logarithmically — the rate of increase is higher when the current transmission rate is much less than the target rate, and slows down as it gets closer to the target rate.

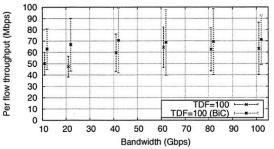
For the network configuration, we use an 80 ms RTT and vary the network bandwidth up to 100 Gbps using underlying 1-Gbps hardware. We configure the machines exactly as in Section 3.3. We perform this experiment for two different protocols: TCP, and TCP with BiC enabled



(a) Validating dilation: TCP performance under dilation matches actual, observed performance.



(b) Using dilation for protocol evaluation: comparing TCP with TCP BiC under high bandwidth.



(c) Pushing the dilation envelope: using a TDF of 100 to evaluate protocols under extremely high bandwidths.

Figure 9: Protocol Evaluation: Per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with an 80-ms RTT as a function of network bandwidth.

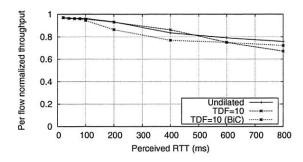


Figure 10: Protocol evaluation: Normalized average per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with 150 Mbps bandwidth as a function of RTT.

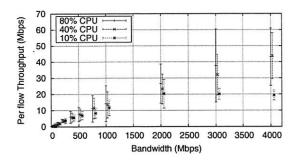


Figure 11: Per-flow throughput of 50 TCP flows across two hosts as a function of network bandwidths. CPU utilization at the sender is restricted to the indicated percentages. Experiments run with TDF of 10.

(henceforth referred to as BiC). In all of the following experiments, we adjust the Linux TCP buffers as suggested in the TCP Tuning Guide [2].

Figure 9 shows per-flow throughput of the 50 connections as a function of network bandwidth. For one execution, we plot the average throughput per flow, and the error bar marks the standard deviation across all flows. In Figure 9(a), the x-axis goes up to 1 Gbps, and represents the regime where the accuracy of time dilation can be validated against actual observations. Figures 9(b) (1 to 10 Gbps) and 9(c) (10 to 100 Gbps) show how time dilation can be used to extrapolate performance.

The graphs show three interesting results. First, time dilation enables us to experiment with protocols beyond hardware limits using implementations rather than simulations. Here we experiment with an unmodified TCP stack beyond the 1 Gbps hardware limit to 100 Gbps. Second, we can experimentally show the impact of high bandwidth-delay products on TCP implementations. Beyond 10 Gbps, per-flow TCP throughput starts to level off. Finally, we can experimentally demonstrate the benefits of new protocol implementations designed for such networks. Figure 9(b) shows that in the 1–10 Gbps regime, BiC outperforms TCP by a significant margin. However, in Figure 9(c) we see that TCP shows a steady, gradual improvement and both BiC and TCP level off beyond 10 Gbps.

TCP performance is also sensitive to RTT. To show this effect under high-bandwidth conditions, we perform another experiment with 50 connections between two machines. However, we instead fix the network bandwidth at 150 Mbps and vary the perceived RTT between the hosts. For clarity, we present an alternative visualization of the results: instead of plotting the absolute per-flow throughput values, we instead plot normalized throughput values as a fraction of maximum potential throughput. For example, with 50 connections on a 150-

Mbps bandwidth link, the maximum average per-flow throughput would be 3 Mbps. Our measured average per-flow throughput was 2.91 Mbps, resulting in a normalized per-flow throughput of 0.97. Figure 10 shows the average per-flow throughput of the three protocols as a function of RTT from 0–800 ms. We chose this configuration to match a recent study on XCP [16], a protocol targeting high bandwidth-delay conditions. The results show the well-known dependence of TCP throughput on RTT, and that the two dilated protocols behave similarly to undilated TCP.

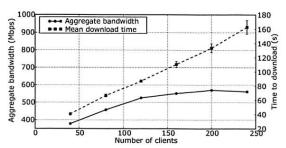
We can also use time dilation as a tool to estimate the computational power required to sustain a target bandwidth. For instance, from Figure 11, we can see that across a 4-Gbps pipe with an 80-ms RTT, 40% CPU on the sender is sufficient for TCP to reach around 50% utilization. This means that processors that are 4 times as fast as today's processors will be needed to achieve similar performance (since 40% CPU at TDF of 10 translates to 400% CPU at TDF of 1).

4.2 High-bandwidth applications

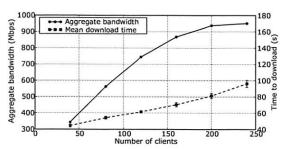
Time dilation can significantly enhance our ability to evaluate data-intensive applications with high bisection bandwidths using limited hardware resources. For instance, the recent popularity of peer-to-peer environments for content distribution and streaming requires significant aggregate bandwidth for realistic evaluations. Capturing the requirements of 10,000 hosts with an average of 1 Mbps per host would require 10 Gbps of emulation capacity and sufficient processing power for accurate study—a hardware configuration that would be prohibitively expensive to create for many research groups.

We show initial results of our ability to scale such experiments using modest hardware configurations with BitTorrent [9], a high-bandwidth peer-to-peer, content distribution protocol. Our goal was to explore the bottlenecks when running a large scale experiment using the publicly available BitTorrent implementation [1] (version 3.4.2).

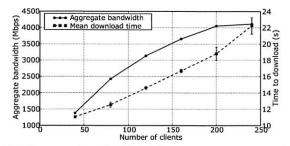
We conducted our experiments using 10 physical machines hosting VMs running BitTorrent clients interconnected through one ModelNet [26] core machine emulating an *unconstrained* network topology of 1,000 nodes. The client machines and the ModelNet core are physically connected via a gigabit switch. The ModelNet topology is unconstrained in the sense that the network emulator forwards packets as fast as possible between endpoints. We create an overlay of BitTorrent clients, all of which are downloading a 46-MB file from an initial "seeder". We vary the number of clients participating in the overlay, distributing them uniformly among the 10 VMs. As a result, the aggregate bisection bandwidth of



(a) VMs are running with TDF of 1 (no dilation). Performance degrades as clients contend for CPU resources.



(b) VMs are running with TDF of 10 and perceived network capacity is 1 Gbps. Dilation removes CPU contention, but network capacity becomes saturated with many clients.



(c) VMs are running with TDF of 10. Perceived network capacity is 10 Gbps. Increasing perceived network capacity removes network bottleneck, enabling aggregate bandwidth to scale until clients again contend for CPU.

Figure 12: Using time dilation for evaluating BitTorrent: Increasing the number of clients results in higher aggregate bandwidths, until the system reaches some bottleneck (CPU or network capacity). Time dilation can be used to push beyond these bottlenecks.

the BitTorrent overlay is limited by the emulation capacity of ModelNet, resource availability at the clients, and the capacity of the underlying hardware.

In the following experiments, we demonstrate how to use time dilation to evaluate BitTorrent performance beyond the physical resource limitations of the test-bed. As a basis, we measure a BitTorrent overlay running on the VMs with a TDF of 1 (no dilation). We scale the number of clients in the overlay from 40 to 240 (4–24 per VM). We measure the average time for downloading the file across all clients, as well as the aggregate bisec-

tion bandwidth of the overlay; we compute aggregate bandwidth as the number of clients times the average per-client bandwidth (file size/average download time). Figure 12(a) shows the mean and standard deviation for 10 runs of this experiment as a function of the number of clients. Since the VMs are not dilated, the aggregate bisection bandwidth cannot exceed the 1-Gbps limit of the physical network. From the graph, though, we see that the overlay does not reach this limit; with 200 clients or more, BitTorrent is able to sustain aggregate bandwidths only up to 570 Mbps. Increasing the number of clients further does not increase aggregate bandwidth because the host CPUs become saturated beyond 20 BitTorrent clients per machine.

In the undilated configuration, CPU becomes a bottleneck before network capacity. Next we use time dilation to scale CPU resources to provide additional processing for the clients without changing the perceived network characteristics. To scale CPU resources, we repeat the previous experiment but with VMs dilated with a TDF of 10. To keep the network capacity the same as before, we restrict the physical capacity of each client host link to 100 Mbps so that the underlying network appears as a 1-Gbps network to the dilated VMs. In effect, we dilate time to produce a new configuration with hosts with 10 times the CPU resources compared with the base configuration, interconnected by an equivalent network. Figure 12(b) shows the results of 10 runs of this experiment. With the increase in CPU resources for the clients, the BitTorrent overlay achieves close to the maximum 1-Gbps aggregate bisection bandwidth of the network. Note that the download times (in the dilated time frame) also improve as a result; due to dilation, though, the experiment takes longer in wall clock time (the most noticeable cost of dilation).

In the second configuration, network capacity now limits BitTorrent throughput. When using time dilation in the second configuration, we constrained the physical links to 100 Mbps so that the network had equivalent performance as the base configuration. In our last experiment, we increase both CPU resources and network capacity to scale the BitTorrent evaluation further. We continue dilating the VMs with a TDF of 10, but now remove the constraints on the network: client host physical links are 1 Gbps again, with a maximum aggregate bisection bandwidth of 10 Gbps in the dilated time frame. In effect, we dilate time to produce a configuration with 10 times the CPU and network resources as the base physical configuration.

Figure 12(c) shows the results of this last experiment. From these results, we see that the "faster" network leads to a significant decrease in download times (in the dilated time frame). Second, beyond 200 clients we see the aggregate bandwidth leveling out, indicating that we

are again running into a bottleneck. On inspection, at that point we find that the end hosts are saturating their CPUs again as with the base configuration. Note, however, that in this case the peak bisection bandwidth exceeds 4 Gbps—performance that cannot be achieved natively with the given hardware.

Based upon these experiments, our results suggest that time dilation is a valuable tool for evaluating large scale distributed systems by creating resource-rich environments. Further exploration with other applications remains future work.

5 Related work

Perhaps the work closest to ours in spirit is SHRiNK [21]. SHRiNK reduces the overhead of simulating large-scale networks by feeding a reduced sample of the original traffic into a smaller-scale network replica. The authors use this technique to predict properties such as the average queueing delays and drop probabilities. They argue that this is possible for TCP-like flows and a network controlled by active queue management schemes such as RED. Compared to this effort, time dilation focuses on speed rather than size and supports unmodified applications.

The idea of changing the flow of time to explore faster networks is not a new one. Network simulators [3, 22, 25] use a similar idea; they run the network in virtual time, independent of wall-clock time. This allows network simulators to explore arbitrarily fast or long network pipes, but the accuracy of the experiments depends on the fidelity of the simulated code to the actual implementation. Complete machine simulators such as SimOS [24] and specialized device simulators such as DiskSim have also been proposed for emulating and evaluating operating systems on future hardware. In contrast, time dilation combines the flexibility to explore future network configurations with the ability to run real-world applications on unmodified operating systems and protocol stacks.

6 Conclusion

Researchers spend a great deal of effort speculating about the impacts of various technology trends. Indeed, the systems community is frequently concerned with questions of scale: what happens to a system when bandwidth increases by X, latency by Y, CPU speed by Z, etc. One challenge to addressing such questions is the cost or availability of emerging hardware technologies. Experimenting at scale with communication or computing technologies that are either not yet available or prohibitively expensive is a significant limitation

to understanding interactions of existing and emerging technologies.

Time dilation enables empirical evaluation at speeds and capacities not currently available from production hardware. In particular, we show that time dilation enables faithful emulation of network links several orders of magnitude greater than physically feasible on commodity hardware. Further, we are able to independently scale CPU and network bandwidth, allowing researchers to experiment with radically new balance points in computation to communication ratios of new technologies.

Acknowledgments

We would like to thank Charles Killian for his assistance with setting up ModelNet experiments, Kashi Vishwanath for volunteering to use time dilation for his research and Stefan Savage and all the anonymous reviewers for their valuable comments and suggestions. Special thanks to our shepherd, Jennifer Rexford, for her feedback and guidance in preparing the camera-ready version. This research was supported in part by the National Science Foundation under CyberTrust Grant No. CNS-0433668 and the UCSD Center for Networked Systems.

References

- [1] http://bittorrent.com.
- [2] Linux TCP tuning guide. http://www-didc.lbl.gov/ TCP-tuning/linux.html. Last accessed 03/25/2006.
- [3] The network simulator ns-2. http://www.isi.edu/ nsnam/ns/. Last accessed 3/13/2006.
- [4] Teragrid. http://www.teragrid.org/. Last accessed 03/25/2006.
- [5] tcpdump/libpcap. http://www.tcpdump.org. Last accessed 03/25/2006.
- [6] AMD. Amd64 secure virtual machine architecture reference manual. http://www.amd.com/us-en/assets/ content_type/white_papers_and_tech_docs/ 33047.pdf. Last accessed 3/13/2006.
- [7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings* of the 19th ACM SOSP (2003), ACM Press, pp. 164–177.
- [8] BUCY, J. S., GANGER, G. R., AND CONTRIBUTORS. The DiskSim Simulation Environment. http://www.pdl.cmu. edu/DiskSim/index.html. Last accessed 3/13/2006.
- [9] COHEN, B. Incentives Build Robustness in BitTorrent. Workshop on Economics of Peer-to-Peer Systems (2003).
- [10] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtualtime (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SOSP* (New York, NY, USA, 1999), ACM Press, pp. 261–276.
- [11] FLOYD, S. High Speed TCP for Large Congestion Windows. http://www.icir.org/floyd/papers/ rfc3649.txt, 2003. RFC 3649.

- [12] FOSTER, I., KESSELMAN, C., NICK, J., AND TUECKE, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus. org/alliance/publications/papers/ogsa.pdf, January 2002. Last accessed 03/29/2006.
- [13] INTEL. Intel virtualization technology. http://www.intel. com/technology/computing/vptech/index.htm. Last accessed 3/13/2006.
- [14] JACOBSON, V., BRADEN, R., AND BORMAN, D. TCP Extensions for High Performance. http://www.rfc-editor.org/rfc/rfc1323.txt, 1992. RFC 1323.
- [15] JAIN, R. The Art of Computer Systems Performance Analysis. John Wiley & Sons, 1991. Chapter 12.
- [16] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. In SIGCOMM (2002), ACM Press, pp. 89–102.
- [17] KELLY, T. Scalable TCP: improving performance in highspeed wide area networks. SIGCOMM Comput. Commun. Rev. 33, 2 (2003), 83–91.
- [18] LAKSHMAN, T. V., AND MADHOW, U. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.* 5, 3 (1997), 336–350.
- [19] LOVE, R. Linux Kernel Development. Novell Press, 2005.
- [20] MOGUL, J. C. TCP offload is a dumb idea whose time has come. In 9th Workshop on Hot Topics in Operating Systems (2003), USENIX.
- [21] PAN, R., PRABHAKAR, B., PSOUNIS, K., AND WISCHIK, D. SHRiNK: A method for scaleable performance prediction and efficient network simulation. In *Proceedings of IEEE INFOCOM* (2003).
- [22] RILEY, G. F. The Georgia Tech Network Simulator. In MoMe-Tools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research (2003), pp. 5–12.
- [23] RIZZO, L. Dummynet: A simple approach to the evaluation of network protocols. SIGCOMM Comput. Commun. Rev. 27, 1 (1997), 31–41.
- [24] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. A. Using the SimOS machine simulator to study complex computer systems. ACM Trans. Model. Comput. Simul. 7, 1 (1997), 78–103.
- [25] SZYMANSKI, B. K., SAIFEE, A., SASTRY, A., LIU, Y., AND MADNANI, K. Genesis: A System for Large-scale Parallel Network Simulation. In Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS) (May 2002).
- [26] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. SIGOPS Oper. Syst. Rev. 36 (2002), 271–284.
- [27] VMWARE. Timekeeping in VMWare Virtual Machines. http://www.vmware.com/pdf/vmware_timekeeping.pdf. Last accessed 03/24/2006.
- [28] WARKHEDE, P., SURIAND, S., AND VARGHESE, G. Fast packet classification for two-dimensional conflict-free filters. In Proceedings of IEEE INFOCOM (July 2001).
- [29] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. SIGOPS Oper. Syst. Rev. 36 (2002), 195–209.
- [30] XU, L., HARFOUSH, K., AND RHEE, I. Binary increase congestion control (BiC) for fast long-distance networks. In *Proceedings of IEEE INFOCOM* (2004).

The Dark Oracle: Perspective-Aware Unused and Unreachable Address Discovery

Evan Cooke, Michael Bailey, Farnam Jahanian
Electrical Engineering and Computer Science Department
University of Michigan
{emcooke, mibailey, farnam}@umich.edu

Richard Mortier Microsoft Research Cambridge, UK mort@microsoft.com

Abstract

Internet traffic destined for unused or unreachable addresses provides critically important information on malicious and misconfigured activity. Since Internet address allocation and policy information is distributed across many devices, applications, and administrative domains, constructing a comprehensive map of unused and unreachable ("dark") addresses is challenging. In this paper, we present an architecture that automates the process of discovering these dark addresses by actively participating with allocation, routing, and policy systems. Our approach is to adopt a local perspective revealing unreachable external addresses and unused private and local addresses, and enabling the detection of threats coming into and out of a network. To validate the approach, we construct a prototype system called the Dark Oracle that uses internal and external routing data and host configuration information, such as DHCP logs, to automatically discover dark addresses. We experimentally evaluate the prototype using data from a large enterprise network, and a regional ISP, and from deployment of the Dark Oracle on a large academic network.

1 Introduction

It was once widely believed that the Internet was in imminent danger of address exhaustion due to millions of new users and the proliferation of new devices. Instead, we now find huge numbers of unused addresses. Large address blocks are still not allocated by registries, blocks allocated to organizations are never externally advertised or routed, and there are millions of unused addresses within allocated and routed subnets between the laptops, desktops, and servers we use every day. During the course of this study we found that 66.8% of all possible IPv4 addresses were never announced through BGP, 57.5% of the addresses assigned to the campus of a large academic network were never internally routed, and 64.8% of the addresses allocated to a DHCP server were never assigned to a host.

This vast pool of unallocated, unrouted, and unassigned addresses sitting idle across the Internet can be used to provide intelligence on malicious and miscon-

figured Internet activity [24]. There are a range of techniques for monitoring contiguous ranges of unused addresses, including honeypots [1, 30, 31], virtual honeypots [3,15,35], emulators [26,37], simple responders [2], and passive packet capture [11, 28]. We refer to these techniques together as *honeynet* monitoring.

Existing honeynet monitoring systems only cover a very small percentage of the available unused address space. Two fundamental problems limit monitoring more addresses. First, address allocation information is distributed across many devices, applications, and administrative domains. For example, address registries like ARIN can provide information on what addresses are assigned to an organization, but not on what addresses are routed or reachable. The second challenge is that address allocations can change quickly. For example, wireless devices can enter and leave a network, and instability in routing information can impact address reachability. The result is that honeynet monitoring systems today monitor only easily obtainable, contiguous blocks of addresses.

This paper presents an architecture that automates the process of discovering these non-productive addresses by participating directly with allocation, routing, and policy systems. The goal is to pervasively discover unused and unreachable ("dark") addresses inside a network so that traffic sent to those addresses can be forwarded to honeynet monitoring systems.

This architecture is fundamentally different from existing systems because it is *perspective-aware*. This means it adopts the local perspective of a specific network, thereby expanding the number of monitorable addresses and enabling *outgoing* honeypots. Today, threats coming into a network [32, 33] receive the most attention; however, threats inside the network, such as infected laptops, are arguably more serious. The proposed architecture discovers addresses that are externally *unreachable* from the perspective of a particular network, and it routes any packets leaving the network that are destined for unreachable addresses to a honeynet. These outgoing monitors provide unique visibility into local infections and misconfigurations.

To demonstrate our approach, we construct the *Dark Oracle*, a system designed to discover unused and unreachable addresses within a network. The system integrates external routing data like BGP, internal routing data like OSPF, and host configuration data like DHCP server logs to construct a locally accurate map of dark addresses. The Dark Oracle automates address discovery, significantly simplifying the process of finding dark addresses. It also provides unique local visibility into internal threats and targeted attacks.

We experimentally evaluate our approach using data from a large enterprise network, and a regional ISP, and from deployment of the Dark Oracle on a large academic network with more than 10,000 hosts. We show how the external, internal, and host configuration address allocation data sources are stable over time, and that the system is scalable. Even when each data source is sampled just once a day, the error in address classification is well under 1%. We deploy a pervasive honeynet detector that uses the addresses from the Dark Oracle and show how unused addresses from a DHCP server reveal almost 80,000 unique source addresses compared to 4,000 found by a traditional /24 monitor. Because we are also able to monitor outgoing addresses, we discover almost 2,000 locally infected or misconfigured hosts in an academic network. These experiments demonstrate the effectiveness of the Dark Oracle in discovering highly distributed local and global dark addresses, thereby enabling quick detection of targeted and internal attacks.

2 Background and Related Work

As Internet-based attacks have become increasingly commonplace and complex, it has become impractical for experts to manually analyze each attack and the hundreds of subsequent variants [9]. This rapid growth in malicious Internet activity has driven the need for more automated data collection and analysis systems.

Approaches to the detection and characterization of network-based threats fall into two general categories: monitoring production systems such as live networks or host-based firewalls [33], and monitoring non-productive *honeypot* resources. This paper focuses on honeypots which provide a unique pre-filtered source of intelligence on the activity of attackers and other anomalous processes [6, 30].

Host-based honeypot systems have traditionally been allocated a single IP address which limits visibility into processes such as random scanning threats [30]. This limitation of monitoring only a single address helped to motivate the development of wide-address monitors called network telescopes [21], sinks [37], blackholes [29], and darknets [11]. These efforts have produced a new understanding of denial of service [22], worms [2, 4, 20, 28], and malicious behavior [24].

Monitoring large numbers of unused addresses simultaneously has been shown to provide quicker and more complete information on threats [8, 16, 17, 21]. Cooke et al. demonstrated that distinct honeynets observed orders-of-magnitude different amounts of traffic and different numbers of unique source IPs [8, 10]. These differences persisted even when accounting for local preference and specific propagation algorithms. Pang et al. also demonstrated that data collected at honeynets at three locations belonging to three distinct networks differed significantly [24]. Kumar et al. recently demonstrated how the Witty worm's random number generator produces non-uniform scanning [17]. However, gathering the same detailed forensic information produced by a real honeypot is a scalability challenge. One approach is to trade fidelity for scalability by emulating operating systems and services rather than running real operating system or application instances [26, 37].

Another approach is to place each honeypot instance within a virtual machine [15, 32]. This enables the execution of multiple operating systems on a single physical machine. Unmodified virtual machines are not sufficiently scalable because a large monitor can receive hundreds or thousands of connections per second. One way of reducing this load is to filter the incoming connections before they reach a honeypot [3]. Another technique is to make the process of storing and spawning virtual machines more efficient. The Potemkin Virtual Honeyfarm [35] uses *copy-on-write* virtual machine images to quickly restore and execute operating system images as packets enter the honeyfarm.

In summary, techniques that monitor unused addresses provide important intelligence on new Internet threats and are becoming more operationally important as Internet-based attacks have become both increasingly commonplace and complex. Recent honeynet scalability advances have provided the framework for monitoring larger and more diverse address ranges and in this paper we attempt to address this need by developing a system designed to pervasively discover these addresses.

3 Redefining Dark Space

When most researchers refer to honeypots, honeynets, darknets, network telescopes, and blackholes there is an implicit assumption that the monitored addresses are globally advertised and globally reachable. That is, a path that exists from most points on the global Internet to the monitored addresses.

This view deserves closer scrutiny. We propose that the number of possible dark addresses would greatly increase if the definition is expanded to include *unreachable* addresses. By adopting the perspective of a particular network, it is possible to discover addresses that may or may not be reachable in other parts of the Internet. A

Botnet Command	Targ. Botnet Command		Targ.	Botnet Command	Targ.	
ipscan r.r.r.r dcom2 -s	No	ipscan i.i.i.i dcom2 -s	No	advscan wkssvcENG 100 0 0	No	
adv.start lsass 198 5 0 -b	No	ipscan s.s.s.s dcom2 -s	No	ipscan r.r.r.r dcom2 -s	No	
ipscan 24.s.s.s dcom	Yes	advscan dcass 300 5 0 141.x.x.x	Yes	advscan lsass 100 5 999 -b	No	
advscan dcass 300 5 0 140.x.x.x	Yes	advscan dcass 300 5 0 140.142.x.x	Yes	ipscan 69.27.s.s dcom2 -s	Yes	
ipscan 207.s.s.s dcom2 -s	Yes	ipscan s.s mssql2000 -s	Yes	ipscan s.s.s lsass -s	Yes	
ipscan 84.9.s.s dcom2 -s	Yes	ipscan s.s webdav3 -s	Yes	ipscan r.r.r.r dcom2 -s	No	
ipscan s.s.s mssql2000 -s	Yes	ipscan 194.s.s.s dcom2 -s	Yes	ipscan 194.116.s.s dcom2	Yes	
advscan lsass_139 50 10 0 128.218.x.x	Yes	ipscan 192.s.s.s dcom2 -s	Yes	ipscan 128.s.s.s dcom2 -s	Yes	

Table 1: Botnet scan commands captured on a live /15 academic network during May 2005. The table shows that 70% of the captured commands were targeted at a specific /8 or /16 network.

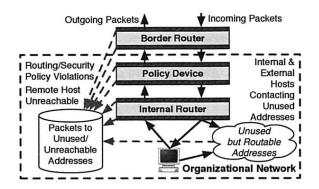


Figure 1: Unused and unreachable addresses inside a network. These addresses can come from a range of sources including routing and policy enforcement devices.

packet leaving a network that would be dropped by an upstream router because the destination address is not allocated is an operationally interesting packet and warrants closer inspection. By locating these upstream and locally unreachable addresses and combining them with unused addresses from throughout the network [13] it is possible to significantly increase the number of dark addresses available to honeynets. Some examples of dark addresses include:

Unused Addresses:

- Unused addresses that are globally advertised and routable
- Unused private addresses that are locally *routable*
- · Unused UDP/TCP ports on an end-system

Unreachable Addresses:

- · Reserved addresses
- · Allocated but unadvertised addresses
- Private addresses that are locally unroutable
- Unused addresses that are globally advertised but *unroutable* (e.g., due to policy)

A pictorial representation of the possible sources of dark addresses is illustrated in Figure 1. Devices and configuration from the routing infrastructure and from policy enforcement mechanisms (e.g., network firewalls) are possible sources of address information. The key idea is that by using a *perpective-aware* address dis-

covery mechanism, it is possible to find and utilize a far greater range of dark addresses.

This broader view of dark addresses provides three fundamental improvements to honeynet systems. First, highly distributed dark addresses enable the detection of targeted attacks and are more difficult to fingerprint. Second, local addresses such as unused private addresses provide a unique perspective into internal threats. Finally, a large number of addresses provides quick detection of randomly propagating threats.

3.1 Perspective-Aware

The expanded definition of dark addresses has implications on how dark addresses are monitored. It is now possible to monitor both *incoming* and *outgoing* traffic. That is, if an address is not internally or externally reachable, that address can be marked as dark. By tracking incoming and outgoing packets, one also gains a unique perspective into local behavior. Below is a list of interesting features one can detect by monitoring incoming and outgoing traffic to dark space.

- Inbound Traffic: Globally-scoped attacks (worms), externally-sourced targeted attacks (botnet scans), backscatter (DOS attacks), and externally-sourced reconnaissance (scans).
- Outbound Traffic: Locally infected machines (worms/botnets), local misconfiguration (misconfigured DNS), and internal reconnaissance (scans).

To explore the importance of having visibility into both incoming and outgoing traffic, we studied the targeting behavior of bot infected computers. Bots have the ability to perform targeted attacks against external hosts and local attacks against internal systems [9]. To investigate the prevalence of targeted bot behavior, we conducted a study of botnet commands. We looked for the specific command signatures of Agobot/Phatbot [7], rBot/SDBot [19], and Ghost-Bot in the payloads of traffic captured in a large academic network. Table 1 shows a list of commands from approximately 11 bots detected by the system during May 2005. Each command instructs the bot to begin scanning a range of IP addresses. We found that 70% of the commands were targeted at external /8 or /16 networks or specified a scan of local systems (e.g., ipscan s.s webdav3 -s). The implica-

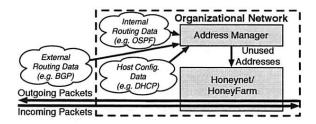


Figure 2: Major components of an automated dark address discovery architecture. Multiple sources of allocation data are used to find *unused* and *unreachable* addresses.

tion is that monitoring targeted attacks is becoming more important, and that distributed, locally-scoped monitoring is critical for obtaining a complete picture of targeted external attacks and internal threats.

In summary, a simple way to dramatically expand the visibility of honeynet systems is to monitor *unreachable* and *unused* addresses. We now describe a system designed to automatically discover these dark addresses inside a network.

4 Architecture

In this section we describe an architecture that automates the process of discovering dark addresses by participating directly with allocation, routing, and policy systems. The architecture is composed of two major components. The first component is the address allocation data sources. There are three main sources of allocation data: external routing data, internal routing data, and host configuration data. The second major architectural component is the address manager that utilizes the address allocation data to provide a map of dark addresses. A high-level diagram that depicts the major components of the architecture is illustrated in Figure 2.

In the next three subsections, we describe possible data sources for the address manager and the importance of using internal data sources. Using this understanding, we develop three classes of allocation data sources that are used as input for the address discovery architecture. Finally, we describe how to combine data from different data sources in a coherent manner.

4.1 Potential Address Sources

Discovering dark address space is challenging because address allocation information is distributed across many devices, applications, and administrative domains. This means that there is no single Internet-wide repository of fine-grained address allocation data. The situation is not much better within organizations as operators rarely have accurate per-device address allocation records. Thus, the key to obtaining accurate information is to integrate data from many sources of address allo-

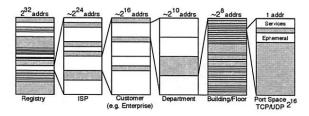


Figure 3: Example of the IPv4/port address allocation hierarchy. The allocation number above each step reflects the relative quantity of addresses being managed in the allocation process.

cation information. To understand where to locate this information, we first need to understand the address allocation process.

To preserve global uniqueness, IP addresses are distributed through a central authority called Internet Assigned Numbers Authority (IANA) [14]. IANA allocates large blocks of address space to regional registries such as ARIN (for North America) that handle address allocations for specific organizations. Certain organizations such as governments and large enterprises also have direct allocations from IANA. Organizations such as ISPs can then turn around and reassign regions of their allocated address space to their customers. For example, an ISP might reassign one or more sub-blocks of addresses to another smaller ISP or enterprise customer.

The addresses used within an organization are often then subdivided by campus, functional unit, or department. A DHCP server is then often used to dynamically allocate addresses to end-hosts. For example, the main site of a large enterprise network might be assigned a /16 and a specific floor within a department might have a DHCP server with the assignment of a /24 address block.

The port allocation process for end-hosts can also be considered part of the address allocation hierarchy. Unlike IP addresses, ports only need to be unique at the host-level so they can be allocated by a host without concern for global uniqueness.

A example of the allocation process is illustrated in Figure 3. The figure also shows the approximate number of addresses being managed at each step. At each step in the allocation process an organization is responsible for uniquely assigning addresses to the next step. For example, an ISP has the responsibility not to assign the same addresses to different customers. Each organization in the process must also only use or advertise addresses assigned to them. The distributed nature of the allocation process means enforcement is a challenge and there are sometimes violations that impact reachablity [18].

Figure 3 also illustrates another important concept. At each step in the allocation process there is often a significant number of unused addresses.

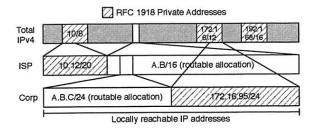


Figure 4: Usage of private address space at different levels in the address allocation hierarchy. Although not illustrated, the private address blocks used at different levels could be from the same address space.

4.2 Leveraging Internal Data

Thus far we have discussed globally unique address allocation which is only part of the process. There are also two other very important classes of dark addresses: private addresses, and policy violations. These addresses provide unique insight into local events. For example, an infected laptop configured with a 192.168.0.0/16 address from a home router is plugged into the network and immediately starts scanning. By monitoring unused portions of private address space, this type of misconfiguration and infection can be quickly identified [10].

Many organizations make extensive use of private address space. An example of private address space usage within an ISP and its customer enterprise is shown in Figure 4. It is difficult to determine the private addresses used within an organization from external data alone. Instead, by using internal routing and host configuration data the unused portions of private address space can be identified. Only small portions of private address space are typically used (10.0.0.0/8 contains 16 million addresses) so visibility into private address space can provide a large number of monitorable local addresses.

Another challenge is that both incoming and outgoing traffic can be blocked by policy applied at different levels in the address allocation hierarchy. Organizations will often use policy to strictly filter incoming traffic or to drop outgoing traffic to certain common ports. For example, an enterprise might block all outgoing TCP port 135 connections to limit outgoing file sharing. If these blocked IP/port pairs can be discovered by communicating with policy systems, then packets to those addresses can instead classified as dark.

Both unused private addresses and policy violations provide a unique source of addresses that are not typically monitored by honeynet systems. The proposed architecture supports the discovery and integration of both of these types of addresses.

4.3 Provisioning the Address Manager

The next step is to determine what data sources should be incorporated into the architecture to provide the broadest possible visibility. As we have argued, the key to discovering the broadest possible range of dark addresses is to take a local perspective. So the question is: What are the data sources available to a particular organization? We argue that there are three broad classes of address allocation data: external routing data, internal routing data, and host configuration data (as illustrated in Figure 2).

External routing data provides information on addresses that have been allocated and are routable. We use routing data rather than data from registries because registry data shows allocation which do not necessarily reflect what address are actually routable. An example source of external routing data is BGP announcements.

Internal routing data is crucial for distributing reachability information inside medium and larger organizations and provides fine-grained information on what addresses are actually allocated within an organization. For example, an ISP may advertise a full /16 through BGP but only half of that space is allocated and used internally for customers. OSPF, ISIS, and RIP are all excellent sources of internal routing data.

Host configuration data includes information from systems that allocate individual addresses to end-hosts. This includes information about address usage like unused ports directly from end-hosts, and configuration from policy devices like firewalls. Host configuration information is available from DHCP and LDAP servers which provide details on specific IP address allocations.

4.4 Synthesizing Allocation Data

Once the external routing data, internal routing data, and host configuration information reaches the address manager – as illustrated in Figure 2 – it must be synthesized into a consistent map of dark addresses. One challenge is how to resolve a conflict when two data sources disagree on the status of an address. For example, external routing data might indicate that an address was reachable while internal routing data reveals it was unused. The solution is to assign priority to the more specific data source. More specific data sources are further down in the allocation hierarchy. For example, host configuration data takes priority over external routing data.

As allocation data from many sources is brought together, it is possible to identify inconsistencies. It is expected that an address that was classified as used by a data source at the top of hierarchy might then be identified as dark by a data source at the bottom. However, if the opposite classification occurs, it can indicate a misconfiguration. For example, if a DHCP server is allocated non-private address block that is not advertised

through BGP, this can indicate that either the server was assigned the wrong addresses or there is a BGP configuration problem.

5 Dark Oracle Design/Implementation

In this section we describe the *Dark Oracle*, the realization of the dark address discovery architecture. The Dark Oracle was implemented in C and Python using a plugin system for different address allocation data sources. The system synthesizes a list of unused addresses based on the address allocation inputs and passes that list of dark addresses to a honeynet.

In the next three subsections, we describe how we constructed the Dark Oracle using BGP external routing advertisements, OSPF internal routing advertisements, and DHCP host configuration data. We then discuss how the addresses from different data sources are combined and how we implemented a prototype honeynet using a promiscuous mode packet sniffer and a high-volume router. Finally, we discuss the issue of misclassified addresses.

5.1 External Routing Data

The source external routing data is BGP, which is the dominant exterior gateway protocol on the Internet to-day. The Dark Oracle obtains an up-to-date view of global BGP announcements using a feed of data from the RouteViews project [34]. RouteViews includes BGP data observed from many vantage points, so it provides a more global view of reachability than a single BGP listener in one network. Depending on the organization and upstream routing policies, it may be important to have more locally-accurate, external reachability information. In this case, it is simple to redirect the BGP module in the Dark Oracle to a local BGP feed.

To determine whether a given IPv4 address is dark, we simply check if there is a valid BGP advertisement for that address. If not, the address is declared dark. Misconfigured BGP advertisements are common across the Internet, so we first filter the advertisements using the bogon list [12].

5.2 Internal Routing Data

To capture internal routing data, the Dark Oracle uses an OSPF listener that participates in the local OSPF backbone and collects update messages [23]. In certain networks, information like router configuration could be helpful to discover details such as static routes, multiple OSPF instances, multiple areas, or other internal routing protocols like RIP. However, this information is not required by the Dark Oracle, it simply improves visibility.

To determine if given address is dark the Dark Oracle must assume the specific perspective of a particular organization. The appropriate address allocation registry, such as ARIN, is checked to decide whether an address is within the range managed by the organization and thus managed by OSPF. If the address falls within the address blocks assigned to the organization, then the current valid OSPF LSA updates are checked to see if the address is advertised. Thus, if an address is allocated to the organization and it is not advertised through OSPF, the address is classified as dark.

There are obvious complications. For example, private address space is potentially valid within an organization, so if a private address is not advertised through OSPF, it is classified as dark. It is also possible that the allocations managed by OSPF are not also assigned through the regional registry. In this case, the Dark Oracle has configuration parameters for managed address ranges.

5.3 Host Configuration Data

The host configuration data source used in the Dark Oracle uses address allocation records from a DHCP server. Rather than modify DHCP server code, the Dark Oracle can passively monitor DHCP commands on the network or directly monitor DHCP logs.

To decide whether a given address is dark, we first need to know if the address falls within the range managed by the DHCP server. To make this decision the DHCP module in the Dark Oracle requires the configured lease time and the pool of addresses from which the DHCP server allocates leases. These parameters are easily extracted from the configuration file or database and can be kept up-to-date with periodic updates. If the address is found to be managed by the DHCP server, we test to see if the address has been allocated by tracking the DHCP discover, lease, and renew messages. If the address has not been allocated, it is declared dark.

5.4 Prioritizing Data Sources

As we outlined in the previous section, the key to combining address allocation data from different sources is to assign priorities. DHCP data has the highest priority, followed by OSPF data and then BGP data. Thus, if DHCP declares an address dark, that assignment takes priority over OSPF or BGP announcements. This process is simple and easily handles additional data sources with different priority levels.

5.5 Prototype Honeynet

Once an address has been classified as dark by the Dark Oracle, that address can then be used for a range of different honeypot applications. One could use a SYN-ACK responder to elicit TCP payloads [2], a system such as *honeyd* to emulate end-host behavior [26], or even forward packets back to a honeyfarm to be executed on real end-hosts [35].

To validate the Dark Oracle we passively captured traffic to the addresses classified as dark. Passive capture is simple, scalable, and provides a large amount of information on malicious activity and misconfiguration [24]. One key piece of information provided by passively captured darknet traffic is the source IP address. The source IP address provides a good estimation of *who* is malicious and misconfigured and doesn't require any honeypot response.

We used two methods for passively capturing traffic: a program called *darktrap*, and a *blackhole* route.

5.5.1 Darktrap

The goal of *darktrap* is to process data from a promiscuous mode interface connected to a span port on a router. A span port mirrors traffic on some or all interfaces of a router to another port. Because this includes live production traffic we also constructed a mechanism to isolate packets to dark addresses.

To deal with large traffic loads (100 to 600 Mb/s), darktrap requires a high-speed evaluation mechanism to indicate whether a given input address is a member of a set of dark addresses. The number of addresses in the dark address pool can also be very large. For example, the BGP table can include almost 200,000 entries.

To obtain the necessary scalability we implemented a hybrid suffix-Patricia tree. Unlike a router which must find a longest-prefix match, darktrap requires a simpler yes/no answer if a prefix exists that covers a given address. The program uses uses a 4-level deep tree for storage in which each tree node is a 256 element-wide array. The tree is populated with the dark prefixes such that each array element in a node is set to either NULL (meaning no match), -1 (meaning a /32 match), or a pointer (meaning a pointer to next level of the tree). darktrap was designed to be integrated into the FreeBSD kernel, but the performance was acceptable in userland. It incurred a few percent CPU overhead on a 3GHz test system, with over 600Mb/s of input traffic using the full set of prefixes from a BGP table dump on September 20, 2005.

5.5.2 Blackhole Routing

The second method we use to capture traffic to dark addresses is a *blackhole* or *fall-through* route. The idea is illustrated in Figure 5. In the example, a network is allocated 1.2.0.0/16 by the RIR, but only advertises 1.2.3.0/24 and 1.2.67.0/24 internally. Thus, the installed blackhole route, 1.2.0.0/16, captures all traffic destined for the network's allocated-but-unrouteable addresses. The idea is similar to adding a route to prevent flooding attacks against persistent loops [36]. The static route identifies all traffic to unused addresses as packets to those addresses fall-through the more specific prefixes

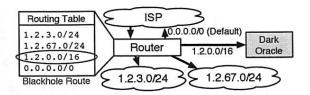


Figure 5: A blackhole route is used to capture traffic that is destined for addresses in the local network that are *not* advertised by any more specific prefix. Traffic destined for external addresses can still be successfully routed by the default route as before.

allocated to live subnets. To collect the traffic, we just placed a monitoring system next to the upstream router and configured the static route to point at the monitoring system.

5.6 Misclassified Addresses

One important problem is misclassified addresses. That is, what if the Dark Oracle misclassifies an address as dark that should be active. There are two main reasons why an address might be misclassified: (1) the state between the Dark Oracle and a data source becomes inconsistent or, (2) there is an inaccuracy in the data source. For example, instability in routing combined with a delay in obtaining routing data could cause inconsistency.

The impact of an address misclassification depends on the monitoring infrastructure and if the honeynets actively respond to incoming packets. For example, misclassifications that occur when using a blackhole route are likely due to operator error and would have happened regardless of a Dark Oracle deployment. However, if a system like *darktrap* is being used, a contention between live systems and honeypot systems can arise. If the address of a server is misclassified, then it is possible that a valid client could interact with a honeypot instead.

The simplest way to avoid misclassification is to minimize inconsistent state and inaccurate data sources. For example, by peering directly with border routers it is possible to minimize inconsistent state between the Dark Oracle and BGP data sources. Inaccurate data sources are often a result of misinformation so education and enforcing strict network policy can minimize inaccuracy.

Despite the best prevention efforts it is still possible to get misclassification. Two steps to reduce the impact are whitelists and less aggressive monitoring. It is possible avoid interactions between legitimate clients and honeypots by whitelisting critical servers. Another technique is to use less aggressive honeynets. For example, a passive capture system that is not inline with the network can be used instead of interactive honeypots for important subnets. Such a system prevents disruption to connectivity but still allows the collection of detailed data.

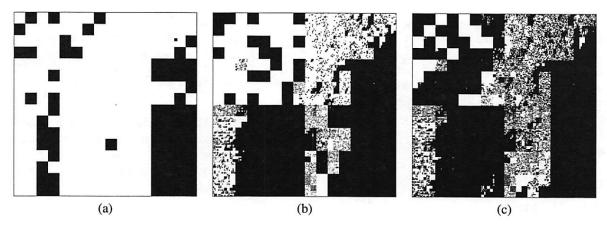


Figure 6: 2D visualization of IP address blocks in the (a) bogon list, (b) allocated by RIPE and ARIN, and (c) advertised via BGP as observed by all RouteViews peers on September 20, 2005. White space represents valid addresses and black space dark addresses and area is proportional to the amount of address space.

6 Dark Oracle Evaluation

In this section we evaluate the proposed architecture and the Dark Oracle prototype. The evaluation is divided into the three parts. In the first part, we use data from a regional ISP, a large enterprise, and an academic network to analyze the quantity, density, and stability of addresses produced by the external routing, internal routing, and host configuration data sources. In the second part, we deploy the *darktrap* and a *blackhole* route on a live network and evaluate the visibility provided by the Dark Oracle by comparing it with existing darknets. Finally, we analyze the effectiveness of using the addresses discovered by the Dark Oracle for detecting targeted and internal attacks.

6.1 Data Source Evaluation

In this subsection we analyze the addresses provided by the different data sources used in the Dark Oracle.

6.1.1 External Routing: BGP

We begin by comparing the BGP data source to similar sources of global Internet reachability information and investigate the stability of the addresses discovered over time. We use address allocations from the major regional registries and non-routable addresses from the bogon list [12] as two other major sources of Internet reachability data. To compare data sources we plotted a snapshot of the prefixes from each data set from September 20, 2005 using a 2D quadrant-based visualization technique that maps all IPv4 space onto a two-dimensional plane [27]. Unused address space is shown in black and used address space in white. Area in the plot is directly proportional to the amount of address space visualized, so a single /8 network takes up 1/256 of the area in each plot. A plot of the bogon list is shown in Figure 6(a); the allocation databases of the two largest regional reg-

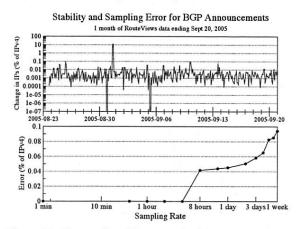


Figure 7: Change in addresses advertised through BGP over time as a percentage of 32-bit (IPv4) space. BGP advertisements observed by all RouteViews peers over one month ending September 20th, 2005.

istries, ARIN and RIPE, are shown in Figure 6(b); and all announced BGP prefixes from RouteViews [34] in Figure 6(c).

Figure 6 shows how allocation information becomes successively more fine-grained as one moves down the allocation hierarchy. The figure also shows qualitatively how the more detailed information provided by the registries and then BGP reveal highly distributed dark addresses. Quantitatively, BGP also reveals the most dark addresses. The bogon list indicates 1,898,557,675 dark addresses, the combined regional registry data reveals 2,396,409,621 dark addresses, and the BGP data reveals 2,872,949,395 dark addresses.

Another question is the stability of the BGP data. That is, how often are addresses added or removed. Churn in BGP announcements is well-documented and although there are often a large number of update messages, we found that the relative amount of addresses that change

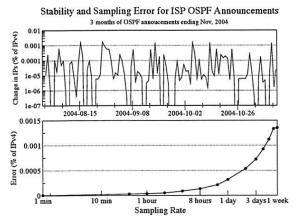


Figure 8: Absolute change in number of addresses advertised through OSPF over 3 months in late 2004 as a percentage of 32-bit (IPv4) space. Observed on the OSPF backbone of a regional ISP.

is quite small. Figure 7 plots the absolute number of addresses that change as a percentage of all possible IPv4 space. We found address churn for BGP is typically between 0.01 and 0.001 percent of all IPv4 space (that's approximately a /16 in size) per 4-hour period.

We also evaluated the error incurred when sampling the BGP data sources. The sampling error is shown in Figure 7. Because the data from RouteViews is updated on a 4-hour basis, the error is zero up to 4 hours. The error with an 8 hour sampling period is 0.04%, which suggests the BGP data source should be updated more frequently. For example, having the BGP module in the Dark Oracle peer directly with the border routers would provide more accurate external reachability information.

6.1.2 Internal Routing: OSPF

To evaluate the use of IGP data for the Dark Oracle, we analyzed OSPF data captured at a large enterprise and a regional service provider. The large enterprise was allocated approximately 900,000 addresses by a regional registry, accounting for 0.02% of all IPv4 space. By analyzing the link state advertisements, we were able discover the number of addresses that were internally routable in a certain part of the network. Over a three-week observation period we discovered 112,423 addresses advertised through OSPF. Of these, 56,139 addresses were from private address space and 56,284 addresses were allocated by a regional registry.

The use of private address space in the enterprise is also very interesting. The 56,139 private addresses were from all three private prefixes (i.e., 10.0.0.0/8, 192.168.0.0/16, 172.16.0.0/12) but only covered 0.3% of the total possible private addresses. This means a huge number of unused private addresses were available.

The mix of addresses observed through OSPF in the regional service provider was somewhat different from

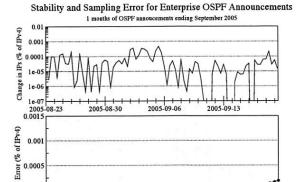


Figure 9: Absolute change in number of addresses advertised through OSPF over 1 month in September 2005 as a percentage of 32-bit (IPv4) space. Observed on the OSPF backbone of a large enterprise network.

Sampling Rate

the enterprise. We observed 20,055,568 allocated and globally routable addresses, which is 0.47% of IPv4 space. Although this is much larger than the enterprise, only 512 addresses from private address space were advertised. This difference may stem from the operational goals of a provider and an enterprise. An enterprise primarily needs IP addresses for local reachability, especially when you consider the widespread use of proxies. On the other hand, a service provider, like the one we profiled, provides global Internet connectivity and thus globally reachable addresses are most important. These differences suggest that a service provider should consider constructing honeynets primarily from globally reachable addresses and an enterprise from large numbers of private addresses.

The addresses advertised through OSPF at the large enterprise and the service provider also showed good stability. Figure 8 shows the address churn at the regional service provider and Figure 9 shows the churn at the large enterprise. The average churn is approximately 0.00001% of IPv4 space per 8 hours. We also measured the error incurred by sampling the data source at different intervals. It turned out much of of the churn was due to the advertisement and withdrawal of a single /32 prefix so the sampling error remained small. Sampling at one-hour intervals produced very little error, so if the Dark Oracle was using OSPF data to interpret the passive output of a blackhole route it could poll the routers instead of participating in OSPF.

6.1.3 Host Configuration: DHCP

To evaluate the utility of the host configuration data source in the Dark Oracle we analyzed the number and stability of dark addresses provided by a DHCP server. We used data from a DHCP server deployed in a department in a large academic network. The DHCP server Fraction of IPs allocated to a DHCP server never assigned to a host 22 subnets (/24s) in academic dept., 1739 allocated hosts

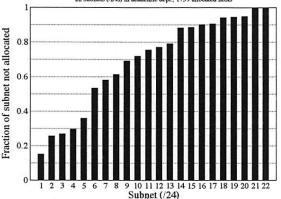


Figure 10: Amount of address space allocated to a DHCP server by /24 subnet in a department of a large academic organization that was never assigned to a host during September, 2005. In total, 70% of the addresses allocated to the DHCP server were never used.

configuration file included 1,802 static entries to allocate addresses based on MAC address.

The DHCP server was assigned a total of 22 /24 subnets from which the 1802 hosts were allocated an IP address. This means that 3319 addresses were never used. The distribution of these unused addresses by subnet is shown in Figure 10. 16 of the 22 subnets were more than 50% unused leaving a large number of dark addresses. Equally interesting, the subnet with the most hosts was still left with 15% of the space unallocated.

We also tracked the amount of time each host was active by monitoring when hosts were assigned or renewed a DHCP leases from the server. Figure 11 shows the number of addresses used over two months. Surprisingly, only about 35% of the 1,802 addresses were in use at any time and the usage was very stable (the DHCP server was configured with a 1-week lease time which likely improved stability). To put this in context, if we were to just use OSPF data, we would observe the 22 subnets allocated to DHCP and assume all 22 were used. But, by using host configuration data we were able to discover that only about 631 addresses out of the possible 5,566 usable addresses were in use.

We also looked at the sampling error incurred by updating the DHCP data source less frequently. As shown in Figure 11, the mean sampling error remains well under 1% for almost 3 days. This is partially related to the long lease time (one week), but also indicates the Dark Oracle could sample much less frequently and maintain almost perfectly in-sync.

Finally, one might expect some hosts connecting through DHCP to come and go with high frequency. We also analyzed how long an address that was newly clasStability and Sampling Error for Unique Hosts at a DHCP Server 1739 hosts in DHCP config deployed in academic dept. (1 week DHCP lease time)

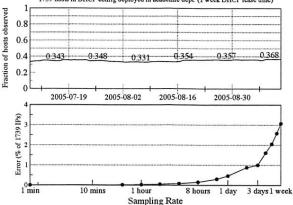


Figure 11: Fraction of unique hosts in the DHCP server configuration file that obtained or renewed an address over time. The average number of hosts active at any one time is approximately 35%. Data over two months in a department at a large academic organization during September, 2005. The DHCP server was configured with 1 week lease times.

sified as dark stayed dark. Over the entire evaluation period we found the mean time an address was classified as dark was 8.85 days and the median was 18.02 days. Thus, a newly dark address will typically stay dark for at least two weeks, although certain addresses fluctuate more rapidly (perhaps due to mobile users).

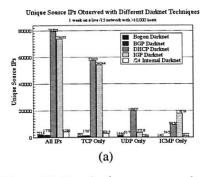
6.2 Live Deployment Results

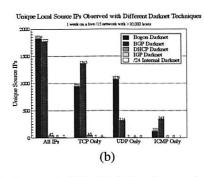
In this subsection we evaluate a live deployment of the Dark Oracle on a real network in a large academic institution. The system was deployed at a central campus router serving approximately 10,000 unique hosts in two /16 networks.

To redirect traffic to our honeynet, we used the *darktrap* program and routing blackhole described earlier. *darktrap* was used to capture traffic to dark addresses discovered by the BGP and host configuration modules, and a routing blackhole was used to capture dark addresses in OSPF. *darktrap* was executed on a 3Ghz system and input traffic was from an optical tap from a span port off a Cisco Catalyst 6500. Traffic destined to the routing blackhole was forwarded to an interface on the same box and integrated with the dark traffic.

6.2.1 Addresses Discovered

Before looking at what was detected, we review the number of dark addresses discovered by the Dark Oracle deployment. The number of prefixes, dark addresses, and total fraction of IP space that was dark for each data source is shown in Figure 12. The fraction of address space that was dark for each data source was computed





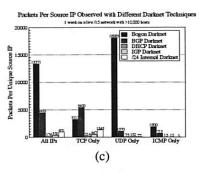


Figure 13: Results from a one week deployment of the Dark Oracle on a large academic network serving approximately 10,000 hosts. Each graph shows the result from the BGP, IGP, host configuration Dark Oracle components. A single /24 darknet is provided for comparison with traditional honeynet monitoring approaches. (a) shows the number of unique source IPs detected, (b) shows the number of unique IPs from within the academic institution address space detected, and (c) shows the number packets per unique source IP.

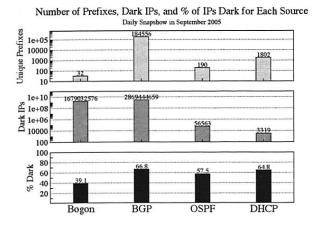


Figure 12: The number of prefixes, dark addresses, and total fraction of IP space dark captured with a snapshot of each Dark Oracle data source on a day in September 2005 in a large academic network.

by taking the number of unused addresses over the total number of addresses managed by the data source. For example, there were 5,120 total addresses allocated to the DHCP server and out of those, 1,801 addresses were configured to be used by the server. An interesting result shown in Figure 12 is that more than 50% of the addresses in the external and internal routing and host configuration sources were dark.

6.2.2 Honeynet Detection Results

To evaluate the utility of the addresses discovered by the Dark Oracle we now characterize the traffic captured by *darktrap* and the blackhole route. The metric we use is the number of unique source addresses observed. The number of unique source IPs provides a first-order approximation of the number of unique infected/misconfigured hosts. We make no attempt to separate misconfigured hosts from infected hosts as both provide important information from the perspective of

network operators. Furthermore, existing signaturebased and prevalence-based detection systems can be used to help identify malicious traffic [25].

We now present results from a one week deployment of the Dark Oracle on a large academic network. The results are shown in Figure 13. For comparison, we also include results from a single statically allocated /24 darknet and a darknet composed of only bogon addresses operating during the same time period within the same academic network.

Source IPs: Figure 13(a) shows the number of unique source IPs detected at the dark addresses discovered using different Dark Oracle data sources. The data is separated by IP protocol. UDP source addresses are sometimes spoofed but the source address on TCP packets are most often valid in order complete the handshake.

The huge number of IPs detected by the IGP and host configuration data sources indicates the importance of having both breadth and good placement. The DHCP data source observed almost 13 times more addresses than the single /24 darknets. Recall that the IGP and host configuration data sources can capture attacks coming into the network. Thus, the almost 80,000 source IPs detected are likely externally-sourced attacks coming into the network. In contrast, the few thousand IPs detected by the bogon and BGP data sources are likely hosts on the same network.

Local Source IPs: To evaluate the locality of the detection results we plotted only those source IPs that were within the address space of the academic network. The results shown in Figure 13(b) indicate that the addresses from the bogon and BGP data sources detected locally infected/misconfigured hosts while the IGP and DHCP data sources revealed external hosts.

Destinations Per Source IP: The bogon and BGP data sources provide addresses for *outgoing* honeynets and thus information on infected/misconfigured hosts from inside the network. However, the bogon and BGP

data sources also reveal many more addresses than the other data sources so we would expect those addresses to capture a higher percentage of the packets from each infected/misconfigured host. Figure 13(c) plots the average number of packets sent by hosts detected with addresses from each data source. As expected, the bogon and BGP data sources provided addresses that have a higher probability of detecting a local host and thus are well-suited for local detection.

6.2.3 Classification Error

To track the number of misclassifications made by the Dark Oracle we wrote a program called *addrmon* that monitored the same router span port as *darktrap* and flagged an IP address as active if it observed that address sending an IP packet. Throughout the entire week-long period we observed 11,118 active IPs on the network. 45 of those IPs were classified as dark by the Dark Oracle (we removed those addresses from our analysis). It is also important to note that we just looked for a single packet so some of those 45 addresses could have been spoofed, and thus were actually dark. Further investigation of those addresses revealed that they were nearly all statically configured hosts.

6.3 Detecting Targeted Attacks

We have shown how the Dark Oracle provides many dark addresses but equally or more important, those addresses are highly distributed throughout the network. We now evaluate how the distributed property of these addresses provides visibility into targeted attacks that would be missed by existing contiguously allocated honeynet systems. Because the addresses are located in many different subnets, honeynet sensors can be pervasively deployed in hundreds or thousands of different parts of the network near to production systems and critical network assets.

To evaluate the importance of having distributed dark addresses we now analyze the time small but well-placed sensors take to detect different targeted attacks. We model an intelligent attacker that has knowledge of which subnets contain vulnerable hosts. Our model is based on botnet scanning behavior which we empirically demonstrated in Section 3. Thus, rather than scanning the entire IPv4 address space the attacker will chose a specific subset like a /24 or /16 to scan.

A random scan of IP space is a straightforward process to model. Previous work has looked at the question of how big a darknet needs to be to detect a random scanning worm with a certain confidence [21]. We can take that understanding and extend it to understand targeted scan detection. Moore *et al.* [21] found that the probability of observing one or more packets from a host with a random scan rate r using a detector with coverage p

Time required to observe with a host (95% conf) with diff scan ranges

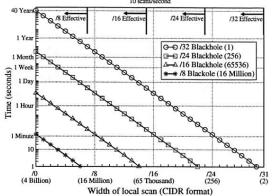


Figure 14: Time required to observe with 95% confidence a packet from a host randomly scanning different ranges of addresses with 4 different sized darknets.

after time T is given by $P(t \le T) = 1 - (1 - p)^{rT}$. They also found that the amount of time T needed to assure a certain probability Z of detecting at least one packet from a scanning host is given by $T = \frac{-1}{r \log_{\frac{1}{2}}(1-p)}$.

In Figure 14, we plot the detection rate with darknets of different sizes as a function of the width of a targeted scan using the above equation. This model the time needed to assure a 95% confidence of detecting a packet from a scan of a certain number of addresses using a darknet having a certain number of addresses. For example, it takes about one minute to detect a packet from a /16 (65,536 addresses) scan with 95% confidence using a /24 (256 addresses) darknet sensor located within the scan range. Detecting a packet from the same scan with the same confidence using a /32 (a single host) would take 5.5 hours. Also, the same local /16 scan could not be detected by a /16 or /8 sensor, which are too large so they are simply not applicable.

The surprising result of this analysis is that even a darknet covering a single address in the right place is an effective tool at detecting targeted scanning behavior. Highly-distributed dark addresses from the Dark Oracle provided by data sources like DHCP and BGP therefore provide the capability to quickly detect targeted incoming and outgoing scans from botnets and other threats.

7 Limitations and Future Work

We wrap up our discussion of the Dark Oracle by discussing possible limitations of the system, describing other novel data sources that could be used to enhance visibility, and detailing how data from different organizations could be combined to construct a powerful, globally-scoped system.

One limitation in deploying a system like the Dark Oracle is the need for access to host configuration data sources. Real networks are complicated and there are often machines that are not in common allocation databases. For example, data centers often have systems with statically configured addresses and many departments manage addresses differently. Informing the Dark Oracle about statically configured machines or getting access to host configuration information in certain parts of the network may not be practical.

Another limitation is address misclassification due to data source instability or inaccuracy. We discussed this issue in Section 5.6 and related several preventive measures to mitigate risk.

There is also the possibility that an attacker could *fingerprint* the dark addresses and attempt to avoid them. Beyond the simple defense of making the honeynets act as much like real systems as possible, the huge range of dark addresses discovered by the Dark Oracle provides strong defense. For example, it is possible to respond with honeypots from IPs that randomly rotate based on the source IP of the attacker. Such simple defenses render algorithms like probe response attacks far more difficult to execute [5]. Even with a complete map of dark addresses, it is impractical to encode them in self-propagating malware like worms due to payload size constraints [38].

The flexibility that makes the Dark Oracle resistant to fingerprinting also makes it very expandable. Because the data sources used in the Dark Oracle are independent, it is simple to deploy the Dark Oracle in stages and add new data sources as needed. There are many data sources that provide allocation data with other interesting perspectives. For example, dark addresses in the address blocks assigned to VPN servers, addresses blocked by network-based and host-based firewalls, and even ACL violations in routers.

One promising pool of dark addresses that could be used with the Dark Oracle is unused TCP and UDP ports. The live computers sitting around a network are often idle and have many unused TCP and UDP ports. A daemon running on each end host could inform the Dark Oracle about these unused ports and packets destined to these unused ports could instead be forwarded to a honeynet.

As a preliminary investigation of the idea of monitoring unused ports we measured the mean number of ports that were used per 5 minutes per local source IP address in the large enterprise and academic network. As Figure 15 shows, there are many unused ports that could be leveraged. Hosts on the academic network used less then 1,000 ports on average which is far less then the possible 65,335 ports. The spikes in the enterprise data are interesting and are likely correlated with backup activity.

Mean Number of TCP Ports Used Per Source IP Per 5 Minutes
9 Days from Large Enterprise, 3 Days from Large Academic Network

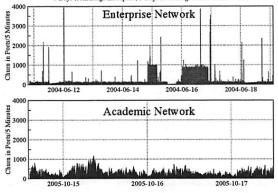


Figure 15: Mean number of TCP port unused per unique local source IP per 5 minutes on a large enterprise and large academic network over a few days. Measurements done in June 2004 and October 2005.

Another interesting research area lies in sharing the allocation data between organizations to improve global visibility. Previous work has looked at sharing dark addresses between an ISP and its customer [16], but it is also possible to connect Dark Oracle instances together to form a global network of fine-grain dark address information services. This would enable organizations to construct much more robust outgoing filtering devices.

8 Conclusion

This paper has introduced the Dark Oracle, a system that automates the process of discovering unused and unreachable addresses inside a network. We described a general architecture that integrates external routing data like BGP, internal routing data like OSPF, and host configuration data like DHCP server logs to construct a locally-accurate map of dark addresses. We experimentally evaluated the Dark Oracle using data from a large enterprise network, a regional ISP, and deployment of the Dark Oracle on a large academic network. We showed how the Dark Oracle provided addresses that revealed almost 80,000 unique source IPs compared to 4,000 with a traditional /24 darknet. We also demonstrated how the unique perspective of Dark Oracle provided visibility into internal threats and targeted attacks. Finally, we described future work and extensions to the Dark Oracle such as leveraging unused TCP and UDP ports on live hosts and combining many Dark Oracles to construct a global dark network.

Acknowledgments

This work was supported by the Department of Homeland Security (DHS) under contract number NBCHC040146, and by corporate gifts from Intel Corporation and Cisco Corporation.

References

- K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.
- [2] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor: A distributed blackhole monitoring system. In *Proceedings of Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, February 2005.
- [3] Michael Bailey, Evan Cooke, Farnam Jahanian, Niels Provos, Karl Rosaen, and David Watson. Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic. Proceedings of the USENIX/ACM Internet Measurement Conference, October 2005.
- [4] Michael Bailey, Evan Cooke, David Watson, Farnam Jahanian, and Jose Nazario. The Blaster Worm: Then and Now. IEEE Security & Privacy, 3(4):26–31, 2005.
- [5] John Bethencourt, Jason Franklin, and Mary Vernon. Mapping Internet sensors with probe response attacks. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.
- [6] Bill Cheswick. An evening with Berferd in which a cracker is lured, endured, and studied. In Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California, pages 163–174, Berkeley, CA, USA, Winter 1992.
- [7] Computer Associates. Win32.Agobot. http://www3.ca.com/ securityadvisor/virusinfo/virus.aspx?id=37776, July 2004
- [8] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, and Farnam Jahanian. Toward understanding distributed blackhole placement. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM-04)*, New York, Oct 2004. ACM Press.
- [9] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie roundup: Understanding, detecting, and disrupting botnets. In Proceedings of the Steps to Reducing Unwanted Traffic on the Internet (SRUTI 2005 Workshop), Cambridge, MA, July 2005.
- [10] Evan Cooke, Z. Morely Mao, and Farnam Jahanian. Hotspots: The root causes of non-uniformity in self-propagating malware. In Proceedings of the International Conference on Dependable Systems and Networks (DSN'2006), June 2006.
- [11] Team Cymru. The darknet project. http://www.cymru.com/ Darknet/index.html, June 2004.
- [12] Team Cymru. The Bogon List. http://www.cymru.com/ Documents/bogon-list.html, June 2005.
- [13] Warren Harrop and Grenville Armitage. Greynets: A definition and evaluation of sparsely populated darknets. In *Proceedings* of the ACM SIGCOMM MineNet Workshop, Philadelphia, PA, August 2005.
- [14] Internet Assigned Numbers Authority (IANA). Internet Protocol V4 Address Space. http://www.iana.org/assignments/ ipv4-address-space, June 2005.
- [15] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, August 2004.
- [16] Balachander Krishnamurthy. Mohonk: Mobile honeypots to trace unwanted traffic early. In Proceedings of the ACM SIG-COMM workshop on Network troubleshooting, pages 277–282. ACM Press, 2004.
- [17] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an internetscale event. Proceedings of the USENIX/ACM Internet Measurement Conference, October 2005.

- [18] Craig Labovitz, Abha Ahuja, and Michael Bailey. Shining Light on Dark Address Space. http://www.arbornetworks.com/, November 2001.
- [19] McAfee. W32/Sdbot.worm. http://vil.nai.com/vil/ content/v_100454.htm, April 2003.
- [20] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. IEEE Security & Privacy, 1(4):33–39, 2003.
- [21] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Network telescopes. Technical Report CS2004-0795, UC San Diego, July 2004.
- [22] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet denial-of-service activity. In *Proceedings of the Tenth USENIX Security Symposium*, pages 9–22, Washington, D.C., August 2001.
- [23] Richard Mortier. Python routeing toolkit. *IEEE Network*, 16(5):3–3, September 2002.
- [24] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, pages 27–40. ACM Press, 2004.
- [25] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks, 31(23-24):2435–2463, 1999.
- [26] Niels Provos. A Virtual Honeypot Framework. In Proceedings of the 13th USENIX Security Symposium, pages 1–14, San Diego, CA, USA, August 2004.
- [27] S. Qiu, Patrick McDaniel, Fabian Monrose, and Avi Rubin. Characterizing address use structure and stability of origin advertizement in interdomain routing. Technical Report NAS-TR-0018-2005, Pennsylvania State University, July 2005.
- [28] Colleen Shannon, David Moore, and Jeffery Brown. Code-Red: a case study on the spread and victims of an Internet worm. In Proceedings of the Internet Measurement Workshop (IMW), December 2002.
- [29] Dug Song, Rob Malan, and Robert Stone. A snapshot of global Internet worm activity. FIRST Conference on Computer Security Incident Handling and Response, June 2002.
- [30] Lance Spitzner. Honeypots: Tracking Hackers. Addison-Wesley, 2002.
- [31] Lance Spitzner et al. The honeynet project. http://project. honeynet.org/, June 2004.
- [32] Symantec Corporation. DeepSight Analyzer. http://analyzer.securityfocus.com/, 2005.
- [33] Johannes Ullrich. DShield. http://www.dshield.org, 2000.
- [34] University of Oregon. RouteViews project. http://www.routeviews.org/.
- [35] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP), Brighton, UK, October 2005.
- [36] Jianhong Xia, Lixin Gao, and Teng Fei. Flooding Attacks by Exploiting Persistent Forwarding Loops. Proceedings of the USENIX/ACM Internet Measurement Conference, October 2005.
- [37] Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the design and use of Internet sinks for network abuse monitoring. In Recent Advances in Intrusion Detection—Proceedings of the 7th International Symposium (RAID 2004), Sophia Antipolis, French Riviera, France, October 2004.
- [38] Cliff C. Zou, Don Towsley, Weibo Gong, and Songlin Cai. Routing worm: A fast, selective attack worm based on IP address information. Umass ECE Technical Report TR-03-CSE-06, November 2003.

Pip: Detecting the Unexpected in Distributed Systems

Patrick Reynolds*, Charles Killian†, Janet L. Wiener‡, Jeffrey C. Mogul‡, Mehul A. Shah‡, and Amin Vahdat†

* Duke University † UC San Diego ‡ HP Labs, Palo Alto

Abstract

Bugs in distributed systems are often hard to find. Many bugs reflect discrepancies between a system's behavior and the programmer's assumptions about that behavior. We present Pip¹, an infrastructure for comparing actual behavior and expected behavior to expose structural errors and performance problems in distributed systems. Pip allows programmers to express, in a declarative language, expectations about the system's communications structure, timing, and resource consumption. Pip includes system instrumentation and annotation tools to log actual system behavior, and visualization and query tools for exploring expected and unexpected behavior². Pip allows a developer to quickly understand and debug both familiar and unfamiliar systems.

We applied Pip to several applications, including FAB, SplitStream, Bullet, and RanSub. We generated most of the instrumentation for all four applications automatically. We found the needed expectations easy to write, starting in each case with automatically generated expectations. Pip found unexpected behavior in each application, and helped to isolate the causes of poor performance and incorrect behavior.

1 Introduction

Distributed systems exhibit more complex behavior than applications running on a single node. For instance, a single logical operation may touch dozens of nodes and send hundreds of messages. Distributed behavior is also more varied, because the placement and order of events can differ from one operation to the next. Bugs in distributed systems are therefore hard to find, because they may affect or depend on many nodes or specific sequences of behavior.

In this paper, we present Pip, a system for automatically checking the behavior of a distributed system against a programmer's expectations about the system. Pip classifies system behaviors as valid or invalid, groups behaviors into sets that can be reasoned about, and presents overall behavior in several forms suited to discovering or verifying the correctness of system behavior.

Bugs in distributed systems can affect structure, performance, or both. A structural bug results in processing or communication happening at the wrong place or in the wrong order. A performance bug results in processing taking too much or too little of any important resource. For example, a request that takes too long may indicate a bottleneck, while a request that finishes too quickly may indicate truncated processing or some other error. Pip supports expressing expectations about both structure and performance and so can find a wide variety of bugs.

We wrote Pip for three broad types of users:

- original developers, verifying or debugging their own system;
- secondary developers, learning about an existing system; and
- system maintainers, monitoring a system for changes.

Our experience shows three major benefits of Pip. First, expectations are a simple and flexible way to express system behavior. Second, automatically checking expectations helps users find bugs that other approaches would not find or would not find as easily. Finally, the combination of expectations and visualization helps programmers explore and learn about unfamiliar systems.

1.1 Context

Programmers employ a variety of techniques for debugging distributed systems. Pip complements existing approaches, targeting different types of systems or different types of bugs. Table 1 shows four approaches and the types of systems or bugs for which they are most useful.

Traditional debuggers and profilers like gdb and gprof are mature and powerful tools for low-level bugs. However, gdb applies to only one node at a time and generally requires execution to be paused for examination. Gprof produces results that can be aggregated offline but has no support for tracing large-scale operations through the network. It is more useful for tuning small blocks of code than distributed algorithms and their emergent behavior.

More recent tools such as Project 5 [1], Magpie [2], and Pinpoint [5] infer causal paths based on traces of net-

Approach	Scenario
gdb and gprof	low-level bugs well illustrated by a single node; core dumps
black boxes	systems with no source-code access, enough self-consistency for statistical inference
model checking	small systems with difficult-to-reproduce bugs
printf	bugs detectable with simple, localized log analyses

Table 1: Other techniques for debugging distributed systems.

work, application, or OS events. Project 5 merely reports inferred behavior, while Magpie and Pinpoint cluster similar behavior and suggest outliers as possible indicators of bugs. Pip also uses causal paths, but instead of relying on statistics and inference, Pip uses explicit path identifiers and programmer-written expectations to gather and check program behavior. We discuss the relationship between Pip and other causal path debugging systems further in Section 6.

Programmers may find some bugs using model checking [10, 16]. Model checking is exhaustive, covering all possible behaviors, while Pip and all the other techniques mentioned above check only the behaviors exhibited in actual runs of the system. However, model checking is expensive and is practically limited to small systems and short runs—often just tens of events. Model checking is often applied to specifications, leaving a system like Pip to check the correctness of the implementation. Finally, unlike model checking, Pip can check performance characteristics.

In practice, the dominant tool for debugging distributed systems has remained unchanged for over twenty years: printf to log files. The programmer analyzes the resulting log files manually or with application-specific validators written in a scripting or string-processing language. In our experience, incautious addition of logging statements generates too many events, effectively burying the few events that indicate or explain actual bugs.

Debugging with log files is feasible when bugs are apparent from a small number of nearby events. If a single invariant is violated, a log file may reveal the violation and a few events that preceded it. However, finding correctness or performance problems in a distributed system of any scale is incredibly labor intensive. In our own experience, it can take days to track down seemingly simple errors. Further, scripts to check log files are brittle because they do not separate the programmer's expectations from the code that checks them, and they must be written anew for each system and for each property being checked.

1.2 Contributions and results

Pip makes the following contributions:

 An expectations language for writing concise, declarative descriptions of the expected behavior of large distributed systems. We present our lan-

- guage design, along with design principles for handling parallelism and for balancing over- and underconstraint of system behavior.
- A set of tools for gathering events, checking behavior, and visualizing valid and invalid behaviors.
- Tools to generate expectations automatically from system traces. These expectations are often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading them.

We applied Pip to several distributed systems, including FAB [25], SplitStream [4], Bullet [13, 15], and Ran-Sub [14]. Pip automatically generated most of the instrumentation for all four applications. We wrote expectations to uncover unexpected behavior, starting in each case from automatically generated expectations. Pip found unexpected behavior in each application and helped to isolate the causes of poor performance and incorrect behavior.

The rest of this paper is organized as follows. Section 2 contains an overview of the Pip architecture and tool chain. Sections 3 and 4 describe in detail the design and implementation of our expectation language and annotation system, respectively. Section 5 describes our results.

2 Architecture

Pip traces the behavior of a running application, checks that behavior against programmer expectations, and displays the resulting valid and invalid behavior in a GUI using several different visualizations.

2.1 Behavior model

We define a model of application behavior for use with Pip. This model does not cover every possible application, but we found it natural for the systems we analyzed.

The basic unit of application behavior in Pip is a path instance. Path instances are often causal and are often in response to an outside input such as a user request. A path instance includes events on one or more hosts and can include events that occur in parallel. In a distributed file system, a path instance might be a block read, a write, or a data migration. In a three-tier web service, path instances might occur in response to user requests. Pip allows the programmer to define paths in whatever way is appropriate for the system being debugged.

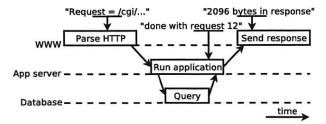


Figure 1: A sample causal path from a three-tier system.

Each path instance is an ordered series of timestamped events. The Pip model defines three types of events: tasks, messages, and notices. A task is like a profiled procedure call: an interval of processing with a beginning and an end, and measurements of resources consumed. Tasks may nest inside other tasks but otherwise may not overlap other tasks on the same thread. Tasks may include asynchronous events like timer callbacks, which Pip normally associates with the path instances that scheduled them. A message is any communication event between hosts or threads, whether a network message, a lock, or a timer. Pip records messages when they are sent and again when they are received. Finally, a notice is an opaque string—like a log message, with a timestamp and a path identifier for context.

Figure 1 shows a sample path instance. Each dashed horizontal line indicates one host, with time proceeding to the right. The boxes are tasks, which run on a single host from a start time to an end time. The diagonal arrows are messages sent from one host to another. The labels in quotation marks are notices, which occur at one instant on a host.

Pip associates each recorded event with a thread. An event-handling system that dispatches related events to several different threads will be treated as having one logical thread. Thus, two path instances that differ only on which threads they are dispatched will appear to have identical behavior.

Our choice of tasks, messages, and notices is well suited to a wide range of distributed applications. Tasks correspond to subroutines that do significant processing. In an event-based system, tasks can correspond to event-handling routines. Messages correspond to network communication, locks, and timers. Notices capture many other types of decisions or events an application might wish to record.

2.2 Tool chain

Pip is a suite of programs that work together to gather, check, and display the behavior of distributed systems. Figure 2 shows the workflow for a programmer using Pip. Each step is described in more detail below.

Annotated applications: Programs linked against Pip's annotation library generate events and resource

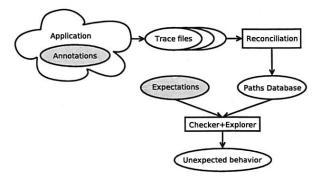


Figure 2: Pip workflow. Shaded ovals represent input that must be at least partially written by the programmer.

measurements as they run. Pip logs these events into trace files, one per kernel-level thread on each host. We optimized the annotation library for efficiency and low memory overhead; it performs no analysis while the application is running.

We found that the required annotations are easiest to add when communication, event handling, and logging are handled by specialized components or by a supported middleware library. Such concentration is common in large-scale distributed systems. For applications linked against a supported middleware library, a modified version of the library can generate automatic annotations for every network message, remote procedure call, and network-event handler. Programmers can add more annotations to anything not annotated automatically.

A separate program gathers traces from each host and reconciles them. Reconciliation includes pairing message send and receive events, pairing task start and end events, and performing a few sanity checks. Reconciliation writes events to a database as a series of path instances. Normally, reconciliation is run offline, parsing log files from a short test run. However, Pip may also be run in an online mode, adding paths to the database and checking them as soon as they complete. Section 4 describes annotations and reconciliation in more detail.

Expectations: Programmers write an external description of expected program behavior. The expectations take two forms: *recognizers*, which validate or invalidate individual path instances, and *aggregates*, which assert properties of sets of path instances. Pip can generate initial recognizers automatically, based on recorded program behavior. These generated recognizers serve as a concise, readable description of actual program behavior. Section 3 describes expectations in more detail.

Formally, a set of recognizers in Pip is a grammar, defining valid and invalid sequences of events. In its current form, Pip allows users to define non-deterministic finite-state machines to check a regular grammar. We chose to define a domain-specific language for defining

these grammars because our language more closely mirrors how programmers reason about behavior in their applications. We believe this choice simplifies writing and maintaining expectations.

Expectation checker: If the programmer provides any expectations, Pip checks all traced behavior against them. These checks can be done non-interactively, to generate a list of violations, or they can be incorporated into the behavior explorer (below). Section 3.5 describes the implementation and performance of expectation checking.

The expectation violations that Pip uncovers do not always indicate bugs in the system being tested. Sometimes, the errors are in the expectations or in the annotations. Using Pip entails changing the application, the expectations, and the annotations until no further unexpected behavior is found. Unexpected paths due to incorrect expectations or annotations can loosely be called *false positives*, though they are not due to any incorrect inference by Pip.

Behavior explorer: Pip provides an interactive GUI environment that displays causal structure, communication structure, sets of validated and invalidated paths, and resource graphs for tasks or paths. Even without writing any expectations, programmers can visualize most aspects of application behavior. Pip stores all of its paths in an SQL database so that users can explore and check application behavior in ways that Pip may not support directly. Space constraints prevent us from describing the GUI or the database schema further here.

3 Expectations

Both checking and visualization in Pip start with expectations. Using Pip's declarative expectations language, programmers can describe their intentions about a system's structure, timing, and resource consumption.

3.1 Design considerations

Our goal is to provide a declarative, domain-specific expectations language that is more expressive than general-purpose languages, resulting in expectations that are easier to write and maintain. Programmers using Pip should be able to find more complex bugs with less effort than programmers checking behavior with scripts or programs written in general-purpose languages.

With expressiveness in mind, we present three goals for any expectations language:

 Expectations written in the language must accept all valid paths. One recognizer should be able to accept a whole family of paths—e.g., all read operations in a distributed file system or all CGI page loads in a webserver—even if they vary slightly. In some systems, particularly event-driven systems, the or-

- der of events might vary from one path instance to the next.
- Expectations written in the language must reject as many invalid paths as possible. The language should allow the programmer to be as specific as possible about task placement, event order, and communication patterns, so that any deviations can be categorized as unexpected behavior.
- The language should make simple expectations easy to express.

We designed Pip with several real systems in mind: peer-to-peer systems, multicast protocols, distributed file systems, and three-tier web servers, among others. Pip also draws inspiration from two platforms for building distributed systems: Mace³ [12] and SEDA [27]. The result is that Pip supports thread-oriented systems, event-handling systems, and hybrids. We gave special consideration to event-handling systems that dispatch events to multiple threads in a pool, i.e., for multiprocessors or to allow blocking code in event handlers.

3.2 Approaches to parallelism

The key difficulty in designing an expectations language is expressing parallelism. Parallelism in distributed systems originates from three main sources: hosts, threads, and event handlers. Processing happens in parallel on different hosts or on different threads within the same host, either with or without synchronization. Event-based systems may exhibit additional parallelism if events arrive in an unknown order.

Pip first reduces the parallelism apparent in an application by dividing behavior into paths. Although a path may or may not have internal parallelism, a person writing Pip expectations is shielded from the complexity of matching complex interleavings of many paths at once.

Pip organizes the parallelism within a path into threads. The threads primitive applies whether two threads are on the same host or on different hosts. Pip's expectation language exposes threading by allowing programmers to write *thread patterns*, which recognize the behavior of one or more threads in the same path instance.

Even within a thread, application behavior can be nondeterministic. Applications with multiple sources of events (e.g., timers or network sockets) might not always process events in the same order. Thus, Pip allows programmers to write *futures*, which are sequences of events that happen at any time after their declaration.

One early design for Pip's expectation language treated all events on all hosts as a single, logical thread. There were no thread patterns to match parallel behavior. This paradigm worked well for distributed hash tables (DHTs) and three-tier systems, in which paths are largely linear, with processing across threads or hosts se-

```
// Read3Others is a validating recognizer
validator Read3Others {
    // no voluntary context switches: never block
     limit(VOL_CS, 0);
    // one Client, issues a read request to Coordinator
     thread Client(*, 1) {
          send(Coordinator) limit(SIZE, {=44b}); // exactly 44 bytes
          recv(Coordinator); }
    // one Coordinator, requests blocks from three Peers
     thread Coordinator(*, 1) {
          recv(Client) limit(SIZE, {=44b});
          task("fabrpc::Read") {
               repeat 3 { send(Peer); }
               repeat 2 {
                   recv(Peer);
                   task("quorumrpc::ReadReply"); }
               future { // these statements match events now or later
                   recv(Peer):
                   task("quorumrpc::ReadReply"); } }
          send(Client); }
     // exactly three Peers, respond to Coordinator
     thread Peer(*, 3) {
          recv(Coordinator);
          task("quorumrpc::ReadReq") { send(Coordinator); } } }
// "assert" indicates an aggregate expectation
assert(average(REAL_TIME, Read3Others) < 30ms);</pre>
```

Figure 3: FAB read protocol, expressed as an expectation.

rialized. It worked poorly, however, for multicast protocols, distributed file systems, and other systems where a single path might be active on two hosts or threads at the same time. We tried a split keyword to allow behavior to occur in parallel on multiple threads or hosts, but it was awkward and could not describe systems with varying degrees of parallelism. The current design, using thread patterns and futures, can naturally express a wider variety of distributed systems.

3.3 Expectation language description

Pip defines two types of expectations: recognizers and aggregates. A recognizer is a description of structural and performance behavior. Each recognizer classifies a given path instance as matching, matching with performance violations, or non-matching. Aggregates are assertions about properties of sets of path instances. For example, an aggregate might state that a specific number of path instances must match a given recognizer, or that the average or 95th percentile CPU time consumed by a set of path instances must be below some threshold.

Figure 3 shows a recognizer and an aggregate expectation describing common read events in FAB [25], a distributed block-storage system. The limit statements are optional and are often omitted in real recognizers. They are included here for illustration.

FAB read events have five threads: one client, one I/O coordinator, and three peers storing replicas of the requested block. Because FAB reads follow a quorum protocol, the coordinator sends three read requests but only needs two replies before it can return the block to

```
validator fab_109 {
    thread t_7(*, 1) {
         send(t_9); recv(t_9); }
     thread t_9(*, 1) {
         recv(t_7);
          task("fabrpc::Read") {
              send(t_1);
              send(t_1);
              send(t_1);
              recv(t_1);
              task("quorumrpc::ReadReply");
              task("quorumrpc::ReadReply"); }
          recv(t_1):
          task("quorumrpc::ReadReply"); }
     thread t_1(*, 3) {
          recv(t_9);
          task("quorumrpc::ReadReq") { send(t_9); } } }
```

Figure 4: Automatically generated expectation for the FAB read protocol, from which we derived the expectation in Figure 3.

the client. The final read reply may happen before or after the coordinator sends the newly read block to the client. Figure 4 shows a recognizer generated automatically from a trace of FAB, from which we derived the recognizer in Figure 3.

The recognizer in Figure 3 matches only a 2-of-3 quorum, even though FAB can handle other degrees of replication. Recognizers for other quorum sizes differ only by constants. Similarly, recognizers for other systems might depend on deployment-specific parameters, such as the number of hosts, network latencies, or the desired depth of a multicast tree. In all cases, recognizers for different sizes or speeds vary only by one or a few constants. Pip could be extended to allow parameterized recognizers, which would simplify the maintenance of expectations for systems with multiple, different deployments.

Pip currently provides no easy way to constrain similar behavior. For example, if two loops must execute the same number of times or if communication must go to and from the same host, Pip provides no means to say so. Variables would allow an expectations writer to define one section of behavior in terms of a previously observed section. Variables are also a natural way to implement parameterized recognizers, as described above.

The following sections describe the syntax of recognizers and aggregate expectations.

3.3.1 Recognizers

Each recognizer can be a *validator*, an *invalidator*, or a building block for other expectations. A path instance is considered valid behavior if it matches at least one validator and no invalidators. Ideally, the validators in an expectations file describe *all* expected behavior in a system, so any unmatched path instances imply invalid be-

havior. Invalidators may be used to indicate exceptions to validators, or as a simple way to check for specific bugs that the programmer knows about in advance.

Each recognizer can match either complete path instances or fragments. A *complete recognizer* must describe all behavior in a path instance, while a *fragment recognizer* can match any contiguous part of a path instance. Fragment recognizers are often, but not always, invalidators, recognizing short sequences of events that invalidate an entire path. The validator/invalidator and complete/fragment designations are orthogonal.

A recognizer matches path instances much the same way a regular expression matches character strings. A complete recognizer is similar to a regular expression that is constrained to match entire strings. Pip's recognizers define regular languages, and the expectation checker approximates a finite state machine.

Each recognizer in Pip consists of expectation statements. Each statement can be a literal, matching exactly one event in a path instance; a variant, matching zero or more events in a path instance; a future, matching a block of events now or later; or a limit, constraining resource consumption. What follows is a description of the expectation statements used in Pip. Most of these statements are illustrated in Figure 3.

Thread patterns: Path instances in Pip consist of one or more threads or thread pools, depending on system organization. There must be at least one thread per host participating in the path. All complete (not fragment) recognizers consist of thread patterns, each of which matches threads. A whole path instance matches a recognizer if each thread matches a thread pattern. Pip's syntax for a thread pattern is:

```
thread(where, count) {statements}
```

Where is a hostname, or "*" to match any host. Count is the number of threads allowed to match, or an allowable range. Statements is a block of expectation statements.

Literal statements: Literal expectation statements correspond exactly to the types of path events described in Section 2. The four types of literal expectation statements are task, notice, send, and recv.

A task statement matches a single task event and any nested events in a path instance. The syntax is:

```
task(name) {statements}
```

Name is a string or regular expression to match the task event's name. The optional *statements* block contains zero or more statements to match recursively against the task event's subtasks, notices, and messages.

A notice statement matches a single notice event. Notice statements take a string or regular expression to match against the text of the notice event.

Send and recv statements match the endpoints of a single message event. Both statements take an identifier indicating which thread pattern or which node the message is going to or arriving from.

Variant statements: Variant expectation components specify a fragment that can match zero or more actual events in a path instance. The five types of variant statements are repeat, maybe, xor, any, and include.

A repeat statement indicates that a given block of code will be repeated n times, for n in a given range. The maybe statement is a shortcut for repeat between 0 and 1. The syntax of repeat and maybe is:

```
repeat between low and high { statements }
maybe { statements }
```

An xor statement indicates that exactly one of the stated branches will occur. The syntax of xor is:

```
xor {
    branch: statements
    branch: statements
    ... (any number of branch statements)
}
```

An any statement matches zero or more path events of any type. An any statement is equivalent to ".*" in a regular expression, allowing an expectation writer to avoid explicitly matching a sequence of uninteresting events.

An include statement includes a fragment expectation inline as a macro expansion. The include statement improves readability and reduces the need to copy and paste code.

Futures: Some systems, particularly event-handling systems, can allow the order and number of events to vary from one path instance to the next. Pip accommodates this fact using future statements and optional done statements. The syntax for future and done statements is:

```
future [name] {statements}
done(name);
```

A future statement indicates that the associated block of statements will match contiguously and in order at or after the current point in the path instance. Loosely, a future states that something will happen either now or later. Futures may be nested: when one future encloses another, it means that the outer one must match before the inner one. Futures may also be nested in (or may include) variant statements. Futures are useful for imposing partial ordering of events, including asynchronous events. Specifying several futures in a row indicates a set of events that may finish in any order. The recognizer in Figure 3 uses futures to recognize a 2-of-3 quorum in FAB: two peers must respond immediately, while the third may reply at any later time.

A done statement indicates that events described by a given future statement (identified by its name) must match prior to the point of the done statement. All futures must match by the end of the path instance, with or without a done statement, or else the recognizer does not match the path instance.

Limits: Programmers can express upper and lower limits on the resources that any task, message, or path can consume. Pip defines several metrics, including real time, CPU time, number of context switches, and message size and latency (the only metrics that apply to messages). A limit on the CPU time of a path is evaluated against the sum of the CPU times of all the tasks on that path. A limit on the real time of a path is evaluated based on the time between the first and last events on the path.

Recognizer sets: One recognizer may be defined in terms of other recognizers. For example, recognizer C may be defined as matching any path instance that matches A and does not match B, or the set difference A - B.

3.3.2 Aggregates

Recognizers organize path instances into sets. Aggregate expectations allow programmers to reason about the properties of those sets. Pip defines functions that return properties of sets, including:

- instances returns the number of instances matched by a given recognizer.
- min, max, avg, and stddev return the minimum, maximum, average, and standard deviation of the path instances' consumption of any resource.

Aggregate expectations are assertions defined in terms of these functions. Pip supports common arithmetic and comparative operators, as well as simple functions like logarithms and exponents. For example:

assert(average(CPU_TIME, ReadOperation) < 0.5s);</pre>

This statement is true if the average CPU time consumed by a path instance matching the ReadOperation recognizer is less than 0.5 seconds.

3.4 Avoiding over- and under-constraint

Expectations in Pip must avoid both over- and underconstraint. An over-constrained recognizer may be too strict and reject valid paths, while an under-constrained recognizer may accept invalid paths. Pip provides variant statements—repeats, xor, and futures—to allow the programmer to choose how specific to be in expressing expectations. Programmers should express how the system should behave rather than how it does behave, drawing upper and lower bounds and ordering constraints from actual program design.

Execution order is particularly prone to under- and over-constraint. For components that devote a thread to

each request, asynchronous behavior is rare, and programmers will rarely, if ever, need to use futures. For event-based components, locks and communication order may impose constraints on event order, but there may be ambiguity. To deal with ambiguity, programmers should describe asynchronous tasks as futures. In particular, periodic background events (e.g., a timer callback) may require a future statement inside a repeat block, to allow many occurrences (perhaps an unknown number) at unknown times.

3.5 Implementation

The Pip trace checker operates as a nested loop: for each path instance in the trace, check it against each recognizer in the supplied expectations file.

Pip stores each recognizer as a list of thread patterns. Each thread pattern is a tree, with structure corresponding to the nested blocks in the expectations file. Figure 5 shows a sample expectation and one matching path. This example demonstrates why a greedy matching algorithm is insufficient to check expectations: the greedy algorithm would match Notice C too early and incorrectly return a match failure. Any correct matching algorithm must be able to check all possible sets of events that variants such as maybe and repeat can match.

Pip represents each path instance as a list of threads. Each thread is a tree, with structure corresponding to the hierarchy of tasks and subtasks. When checking a recognizer against a given path instance, Pip tries each thread in the path instance against each thread pattern in the recognizer. The recognizer matches the path instance if each path thread matches at least one thread pattern and each thread pattern matches an appropriate number of path threads.

Each type of expectation statement has a corresponding check function that matches path instance events. Each check function returns each possible number of events it could match. Literal statements (task, notice, send, and recv) match a single event, while variant statements (repeat, xor, and any) can match different numbers of events. For example, if two different branches of an xor statement could match, consuming either two or three events, check returns the set [2, 3]. If a literal statement matches the current path event, check returns [1], otherwise \emptyset . When a check function for a variant statement returns [0], it can be satisfied by matching zero events. A failure is indicated by the empty set, \emptyset .

The possible-match sets returned by each expectation statement form a search tree, with height equal to the number of expectation statements and width dependent on how many variant statements are present in the expectation. Pip uses a depth-first search to explore this search tree, looking for a leaf node that reaches the end

Figure 5: A sample fragment recognizer and a path that matches it.

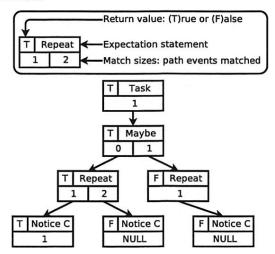


Figure 6: The search tree formed when matching the expectation and the path events in Figure 5.

of the expectation tree and the path tree at the same time. That is, the match succeeds if, in any branch of the search tree, the expectation matches all of the path events.

Figure 6 shows the possibilities searched when matching the expectations and the path events in Figure 5. Each node represents a check function call. Each node shows the return value (true or false) of the recursive search call, the expectation statement being matched, and the number(s) of events it can match. Leaves with no possible matches are shown with a possible-match set of NULL and a return value of false. A leaf with one or more possible matches might still return false, if any path events were left unmatched.

3.5.1 Futures

Pip checks futures within the same framework. Each check function takes an additional parameter containing a table of all currently available futures. Possiblematch sets contain <events matched, futures table> tuples rather than just numbers of events that could be matched. Most check calls do not affect the table of active futures, simply returning the same value passed as a parameter. Future.check inserts a new entry into the futures table but does not attempt to match any events; it returns a single tuple: <0 events, updated futures table>. Done.check forces the named future to match immediately and removes it from the futures table.

Each node in the search tree must try all futures in

Figure 7: The same path as in Figure 5, with a slightly modified recognizer to match it. Note that the notice ("C") statement has been moved into a future block.

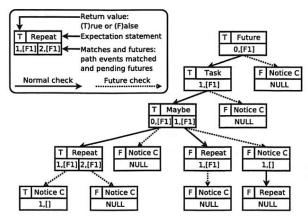


Figure 8: The search tree formed when matching the expectation and the path events in Figure 7.

the table as well as the next expectation statement. If a future matches, then that branch of the tree uses a new futures table with that one future removed. A leaf of the tree matches only if each expectation statement returns success, all path events are consumed, and the futures table is empty.

Figure 7 shows the same path instance as in Figure 5, with a different expectation to match it: the notice("C") statement is now a future. Figure 8 shows the possibilities searched when matching the expectations and the path events in Figure 7. Lazy evaluation again means that only a few nodes of the tree depicted in Figure 8 are actually expanded.

3.5.2 Performance

The time to load and check a path instance depends, of course, on the complexity of the path instance and the complexity of the recognizers Pip checks it against. On a 1.6 GHz laptop running Linux 2.6.13 and MySQL 4.1, a complex path instance containing 100 hosts and 1700 events takes about 12 ms to load and another 12 ms to check against seven recognizers, two of which contain futures. Thus, Pip can load and check about 40 complex path instances, or as many as 3400 simple path instances, per second on this hardware.

4 Annotations

Pip represents behavior as a list of path instances that contain tasks, notices, and messages, as described in Sec-

tion 2. These events are generated by source-code annotations. We chose annotations over other event and tracing approaches for two reasons. First, it was expedient. Our focus is expectations and how to generate, check, and visualize them automatically. Second, most other sources of events do not provide a path ID, making them less detailed and less accurate than annotations. Pip could easily be extended to incorporate any event source that provides path IDs.

Pip provides a library, libannotate, that programmers link into their applications. Programmers insert a modest number of source code annotations indicating which path is being handled at any given time, the beginning and end of interesting tasks, the transmission and receipt of messages, and any logging events relevant to path structure.

The six main annotation calls are:

- annotate_set_path_id(id): Indicate which path all subsequent events belong to. An application must set a path identifier before recording any other events. Path identifiers must be unique across all hosts and all time. Often, identifiers consist of the host address where the path began, plus a local sequence number.
- annotate_start_task(name): Begin some processing task, event handler, or subroutine. Annotation overhead for a task is around 10 μs, and the granularity for most resource measurements is a scheduler time slice. Thus, annotations are most useful for tasks that run for the length of a time slice or longer.
- annotate_end_task(name): End the given processing task.
- annotate_send(id, size): Send a message with the
 given identifier and size. Identifiers must be unique
 across all hosts and all time. Often, identifiers consist of the address of the sender, an indication of
 the type of message, and a local sequence number.
 Send events do not indicate the recipient address,
 allowing logical messages, anycast messages, forwarding, etc.
- annotate_receive(id, size): Receive a message with
 the given identifier and size. The identifier must
 be the same as when the message was sent, usually
 meaning that at least the sequence number must be
 sent in the message.
- annotate_notice(string): Record a log message.

Programs developed using a supported middleware layer may require only a few annotations. For example, we modified Mace [12], a high-level language for building distributed systems, to insert five of the six types of annotations automatically. Our modified mace adds begin- and end-task annotations for each transition (i.e., event handler), message-send and message-receive annotations for each network message and each timer, and set-

path-id annotations before beginning a task or delivering a message. Only notices, which are optional and are the simplest of the six annotations, are left to the programmer. The programmer may choose to add further message, task, and path annotations beyond what our modified Mace generates.

Other middleware layers that handle event handling and network communication could automate annotations similarly. For example, we believe that SEDA [27] and RPC platforms like CORBA could generate message and task events and could propagate path IDs. Pinpoint [5] shows that J2EE can generate network and task events.

4.1 Reconciliation

The Pip annotation library records events in local trace files as the application runs. After the application terminates, the Pip reconciler gathers the files to a central location and loads them into a single database. The reconciler must pair start- and end-task events to make unified task events, and it must pair message-send and message-receive events to make unified message events.

The reconciler detects two types of errors. First, it detects incomplete (i.e., unpaired) tasks and messages. Second, it detects reused message IDs. Both types of errors can stem from annotation mistakes or from application bugs. In our experience, these errors usually indicate an annotation mistake, and they disappear entirely if annotations are added automatically.

5 Results

We applied Pip to several distributed systems, including FAB [25], SplitStream [4], Bullet [13, 15], and Ran-Sub [14]. We found 18 bugs and fixed most of them. Some of the bugs we found affected correctness—for example, some bugs would result in SplitStream nodes not receiving data. Other bugs were pure performance improvements—we found places to improve read latency in FAB by 15% to 50%. Finally, we found correctness errors in SplitStream and RanSub that were masked at the expense of performance. That is, mechanisms intended to recover from node failures were instead recovering from avoidable programming errors. Using Pip, we discovered the underlying errors and eliminated the unnecessary time the protocols were spending in recovery code.

The bugs we found with Pip share two important characteristics. First, they occurred in actual executions of the systems under test. Pip can only check paths that are used in a given execution. Thus, path coverage is an important, though orthogonal, consideration. Second, the bugs manifested themselves through traced events. Program annotations must be comprehensive enough and expectations must be specific enough to isolate unexpected behavior. However, the bugs we found were not limited to bugs we knew about. That is, most of the bugs we

	Lines	Recognizers	Lines of	Number	Number	Trace	Reconciliation	Checking	Bugs	Bugs
System	of code	(lines)	annotations	of hosts	of events	duration	time (sec)	time (sec)	found	fixed
FAB	124,025	17 (590)	28	4	88,054	4 sec	6	7	2	1
SplitStream	2,436	19 (983)	8	100	3,952,592	104 sec	1184	837	13	12
Bullet	2,447	1 (38)	23	100	863,197	71 sec	140	81	2	0
RanSub	1,699	7 (283)	32	100	312,994	602 sec	47	9	2	1

Table 2: System sizes, the effort required to check them, and the number of bugs found and fixed.

found were not visible when just running the applications or casually examining their log files.

Table 2 shows the size of each system we tested, along with how much programmer effort and CPU time it took to apply Pip in each case. Bullet has fewer expectations because we did not write validators for all types of Bullet paths. SplitStream has many expectations because it is inherently complex and because in some cases we wrote both a validator and an overly general recognizer for the same class of behavior (see Section 5.2). Over 90% of the running time of reconciliation and checking is spent in MySQL queries; a more lightweight solution for storing paths could yield dramatic speed improvements. In addition to the manual annotations indicated in the table, we added 55 annotation calls to the Mace compiler and library and 19 to the FAB IDL compiler.

Reconciliation time is $O(E \lg p)$ for E events and p path instances, as each event is stored in a database, indexed by path ID. The number of high-level recognizer checking operations is exactly rp for p path instances and r recognizers. Neither stage's running time is dependent on the number of hosts or on the concurrency between paths. The checking time for a path instance against a recognizer is worst-case exponential in the length of the recognizer, e.g., when a recognizer with pathologically nested future and variant statements almost matches a given path instance. In practice, we did not encounter any recognizers that took more than linear time to check.

In the remainder of this section, we will describe our experiences with each system, some sample bugs we found, and lessons we learned.

5.1 FAB

A Federated Array of Bricks (FAB) [25] is a distributed block storage system built from commodity Linux PCs. FAB replicates data using simple replication or erasure coding and uses majority voting protocols to protect against node failures and network partitions. FAB contains about 125,000 lines of C++ code and a few thousand lines of Python. All of FAB's network code is automatically generated from IDL descriptions written in Python. The C++ portions of FAB combine user-level threading and event-handling techniques. A typical FAB configuration includes four or more hosts, background membership and consensus communication, and a mix of concurrent read and write requests from one or more clients.

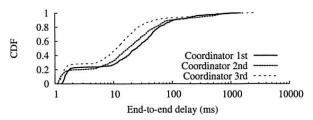


Figure 9: CDF of end-to-end latency in milliseconds for FAB read operations. The left-most line shows the case where the coordinator calls itself last. Note that the x axis is log-scaled to show detail.

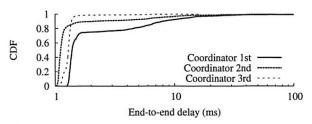


Figure 10: CDF of end-to-end latency in milliseconds for FAB read operations in a system with a high cache hit rate. The left-most line shows the case where the coordinator calls itself second. Note that the x axis is log-scaled to show detail.

We were not initially familiar with FAB, but we had access to its source code, and one of its authors offered to help us understand it. With just a few hours of effort, we annotated FAB's IDL compiler, and were able to get the tasks and messages necessary to examine every protocol.

Figure 3 in Section 3.3 showed an expectation for the FAB read protocol when the node coordinating the access (the I/O coordinator) does not contain a replica of the block requested. In this section, we focus on the case where the coordinator does contain a replica. In addition to the read and write protocols, we annotated and wrote expectations for FAB's implementation of Paxos [17] and the Cristian-Schmuck membership protocol [6] but did not find any bugs in either.

Bugs: When the FAB I/O coordinator contains a replica of the block requested, the order of RPCs issued affects performance. In FAB, an RPC issued by a node to itself is handled synchronously. Originally, FAB issued read or write RPCs to all replicas in an arbitrary order. A recent optimization changed this code so that the coordinator always issues the RPC to itself (if any) last, allowing greater overlap of computation.

FAB's author sent us the unoptimized code without describing the optimization to us, with the intention that we use Pip to rediscover the same optimization. Figure 9 shows the performance of read operations when the coordinator calls itself first, second, or last. When the block is not in cache (all delay values about 10 ms), having the coordinator issue an RPC to itself last is up to twice as fast as either other order. Write performance shows a similar, though less pronounced, difference.

We discovered this optimization using expectations and the visualization GUI together. We wrote recognizers for the cases where the coordinator called itself first, second, and third and then graphed several properties of the three path sets against each other. The graph for end-to-end delay showed a significant discrepancy between the coordinator-last case and the other two cases.

Figure 10 shows the same measurements as Figure 9, in a system with a higher cache hit rate. We noticed that letting the coordinator call itself second resulted in a 15% decrease in latency for reads of cached data by performing the usually unneeded third call after achieving a 2-of-3 quorum and sending a response to the client. The FAB authors were not aware of this difference.

Lessons: Bugs are best noticed by someone who knows the system under test. We wrote expectations for FAB that classified read and write operations as valid regardless of the order of computation. We found it easy to write recognizers for the actual behavior a system exhibits, or even to generate them automatically, but only someone familiar with the system can say whether such recognizers constitute real expectations.

5.2 SplitStream

SplitStream [4] is a high-bandwidth content-streaming system built upon the Scribe multicast protocol [24] and the Pastry DHT [23]. SplitStream sends content in parallel over a "forest" of 16 Scribe trees. At any given time, SplitStream can accommodate nodes joining or leaving, plus 16 concurrent multicast trees. We chose to study SplitStream because it is a complex protocol, we have an implementation in Mace, and our implementation was exhibiting both performance problems and structural bugs. Our SplitStream tests included 100 hosts running under ModelNet [26] for between two and five minutes.

Bugs: We found 13 bugs in SplitStream and fixed most of them. Space does not allow descriptions of all 13 bugs. We found two of the bugs using the GUI and 11 of the bugs by either using or writing expectations. Seven bugs had gone unnoticed or uncorrected for ten months or more, while the other six had been introduced recently along with new features or as a side effect of porting SplitStream from MACEDON to Mace. Four of the bugs we found were due to an incorrect or incomplete under-

standing of the SplitStream protocol, and the other nine were implementation errors. At least four of the bugs resulted in inefficient (rather than incorrect) behavior. In these cases, Pip enabled performance improvements by uncovering bugs that might have gone undetected in a simple check of correctness.

One bug in SplitStream occurred twice, with similar symptoms but two different causes. SplitStream allows each node to have up to 18 children, but in some cases was accepting as many as 25. We first discovered this bug using the GUI: visualizations of multicast paths' causal structure sometimes showed nodes with too many children. The cause the first time was the use of global and local variables with the same name; SplitStream was passing the wrong variable to a call intended to offload excess children. After fixing this bug, we wrote a validator to check the number of children, and it soon caught more violations. The second cause was an unregistered callback. SplitStream contains a function to accept or reject new children, but the function was never called.

Lessons: Some bugs that look like structural bugs affect only performance, not correctness. For example, when a SplitStream node has too many children, the tree still delivers data, but at lower speeds. The line between structural bugs and performance bugs is not always clear.

The expectations checker can help find bugs in several ways. First, if we have an expectation we know to be correct, the checker can flag paths that contain incorrect behavior. Second, we can generate recognizers automatically to match existing paths. In this case, the recognizer is an external description of actual behavior rather than expected behavior. The recognizer is often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading it. Finally, we can write an overly general recognizer that matches all multicast paths and a stricter, validating recognizer that matches only correct multicast paths. Then we can study incorrect multicast paths—those matched by the first but not the second—without attempting to write validators for other types of paths in the system.

5.3 Bullet

Bullet [13, 15] is a third-generation contentdistribution mesh. Unlike overlay multicast protocols, Bullet forms a mesh by letting each downloading node choose several peers, which it will send data to and receive data from. Peers send each other lists of which blocks they have already received. One node can decide to send (push) a list of available blocks to its peers, or the second can request (pull) the list. Lists are transmitted as deltas containing only changes since the last transmission between the given pair of nodes.

Bugs: We found two bugs in Bullet, both of which are inefficiencies rather than correctness problems. First, a

given node A sometimes notifies node B of an available block N several times. These extra notifications are unexpected behavior. We found these extra notifications using the reconciler rather than the expectations checker. We set each message ID as <sender, recipient, block number> instead of using sequence numbers. Thus, whenever a block notification is re-sent, the reconciler generates a "reused message ID" error.

The second bug is that each node tells each of its peers about every available block, even blocks that the peers have already retrieved. This bug is actually expected behavior, but in writing expectations for Pip we realized it was inefficient.

Lessons: We were interested in how notifications about each block propagate through the mesh. Because some notifications are pulls caused by timers, the propagation path is not causal. Thus, we had to write additional annotations for *virtual* paths in addition to the causal paths that Mace annotated automatically.

Pip can find application errors using the reconciler, not just using the path checker or the GUI. It would have been easy to write expectations asserting that no node learns about the same block from the same peer twice, but it was not necessary because the reconciler flagged such repeated notifications as reused message IDs.

5.4 RanSub

RanSub [14] is a building block for higher-level protocols. It constructs a tree and tells each node in the tree about a uniformly random subset of the other nodes in the tree. RanSub periodically performs two phases of communication: *distribute* and *collect*. In the distribute phase, each node starting with the root sends a random subset to each of its children. In the collect phase, each node starting with the leaves sends a summary of its state to its parent. Interior nodes send a summary message only after receiving a message from all children. Our RanSub tests involved 100 hosts and ran for 5 minutes.

Because RanSub is written in Mace, we were able to generate all needed annotations automatically.

Bugs: We found two bugs in RanSub and fixed one of them. First, each interior node should only send a summary message to its parent after hearing from all of its children. Instead, the first time the collect phase ran, each interior node sent a summary message after hearing from one child. We found this bug by writing an expectation for the collect-and-distribute path; the first round of communication did not match. The root cause was that interior nodes had some state variables that did not get initialized until after the first communication round. We fixed this bug.

The second bug we found in RanSub is a performance bug. The end-to-end latency for collect-and-distribute

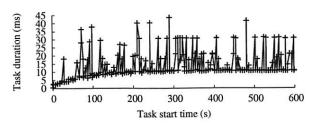


Figure 11: Duration for the deliverGossip task as a function of time.

paths starts out at about 40 ms and degrades gradually to about 50 ms after running for three minutes. We traced the bottleneck to a task called deliverGossip that initially takes 0 ms to run and degrades gradually to about 11 ms. We found this bug using the GUI. First, we examined the end-to-end latency as a function of time. Seeing an error there, we checked each class of tasks in turn until we found the gossip task responsible for the degradation. Figure 11 shows the time consumed by the gossip task as a function of time. The reason for deliverGossip degrading over time is unclear but might be that deliverGossip logs a list of all gossip previously received.

6 Related work

Pip is one of many approaches to finding structure and performance bugs in distributed systems. Below, we highlight two categories of debugging approaches: path analysis tools and automated expectation checking. Pip is the first to combine the two approaches. Finally, we discuss the relationship between Pip and high-level languages for specifying and developing distributed systems.

6.1 Path analysis tools

Several previous systems have modeled the behavior of distributed systems as a collection of causal paths. This approach is particularly appropriate for systems driven by user requests, as it captures the delays and resource consumption associated with each request. Pathbased debugging can enable programmers to find aberrant paths and to optimize both throughput and end-to-end latency.

Project 5 [1] infers causal paths from black-box network traces. By doing so, it can help debug systems with unavailable source code. Deploying black-box debugging, at least in theory, requires less effort than annotating source code. However, Project 5 can only report what it can infer. Its granularity is limited to host-to-host communication, and it often reconstructs paths incorrectly. In particular, interesting paths, including infrequent paths or paths with long or variable delays, may be lost.

Magpie [2] reconstructs causal paths based on OSlevel event tracing. Like Project 5, Magpie can operate without access to source code. However, Magpie can construct paths with much higher accuracy than Project 5 can, because OS-level tracing provides more information than network tracing alone. Magpie clusters causal paths using a string-edit-distance algorithm and identifies outliers—that is, small clusters.

Like Pip, Pinpoint [5] constructs causal paths by annotating applications or platforms to generate events and maintain a unique path identifier per incoming request. Like Pip and Magpie, Pinpoint can construct paths with a high degree of confidence because it does not rely on inference. Like Magpie but unlike Pip, Pinpoint assumes that anomalies indicate bugs. Pinpoint uses a probabilistic, context-free grammar to detect anomalies on a perevent basis rather than considering whole paths. Doing so significantly underconstrains path checking, which, as the authors point out, may cause Pinpoint to validate some paths with bugs.

All three of these existing causal path debugging systems rely on statistical inference to find unusual behavior and assume that unusual behavior indicates bugs. Doing so has two drawbacks. First, inference requires large traces with many path instances. Second, these systems can all miss bugs in common paths or incorrectly identify rare but valid paths.

The accuracy and granularity of existing causal path debugging tools are limited by what information they can get from traces of unmodified applications. In practice, these systems entail a form of gray-box debugging, leveraging prior algorithmic knowledge, observations, and inferences to learn about the internals of an unmodifiable distributed system. In contrast, Pip assumes the ability to modify the source for at least parts of a distributed system, and it provides richer capabilities for exploring systems without prior knowledge and for automatically checking systems against high-level expectations.

6.2 Automated expectation checking

Several existing systems support expressing and checking expectations about structure or performance. Some of the systems operate on traces, others on specifications, and still others on source code. Some support checking performance, others structure, and others both. Some, but not all, support distributed systems.

PSpec [21] allows programmers to write assertions about the performance of systems. PSpec gathers information from application logs and runs after the application has finished running. The assertions in PSpec all pertain to the performance or timing of intervals, where an interval is defined by two events (a start and an end) in the log. PSpec has no support for causal paths or for application structure in general.

Meta-level compilation (MC) [8] checks source code for static bugs using a compiler extended with systemspecific rules. MC checks all code paths exhaustively but is limited to single-node bugs that do not depend on dynamic state. MC works well for finding the root causes of bugs directly, while Pip detects symptoms and highlights code components that might be at fault. MC focuses on individual incorrect statements, while Pip focuses on the correctness of causal paths, often spanning multiple nodes. MC finds many false positives and bugs with no effect, while Pip is limited to actual bugs present in a given execution of the application.

Paradyn [19] is a performance measurement tool for complex parallel and distributed software. The Paradyn Configuration Language (PCL) allows programmers to describe expected characteristics of applications and platforms, and in particular to describe metrics; PCL seems somewhat analogous to Pip's expectation language. However, PCL cannot express the causal path structure of threads, tasks and messages in a program, nor does Paradyn reveal the program's structure.

6.3 Domain-specific languages

Developers of distributed systems have a wide variety of specification and implementation languages to choose from. Languages like Estelle [11], π -calculus [20], join-calculus [9], and P2 [18] embrace a formal, declarative approach. Erlang [3] and Mace [12] use an imperative approach, with libraries and language constructs specialized for concurrency and communication. Finally, many programmers still use traditional, general-purpose languages like Java and C++.

Pip is intended primarily for developers using imperative languages, including both general-purpose languages and domain-specific languages for building distributed systems. Pip provides language bindings for Java, C, C++, and Mace. While programmers using declarative languages can verify the correctness of their programs through static analysis, Pip is still valuable for monitoring and checking dynamic properties of a program, such as latency, throughput, concurrency, and node failure.

7 Conclusions

Pip helps programmers find bugs in distributed systems by comparing actual system behavior to the programmer's expectations about that behavior. Pip provides visualization of expected and actual behavior, allowing programmers to examine behavior that violates their expressed expectations, and to search interactively for additional unexpected behavior. The same techniques can help programmers learn about an unfamiliar system or monitor a deployed system.

Pip can often generate any needed annotations automatically, for applications constructed using a supported middleware layer. Pip can also generate initial expectations automatically. These generated expectations are

often the most readable description of system behavior, and bugs can be obvious just from reading them.

We applied Pip to a variety of distributed systems, large and small, and found bugs in each system.

References

- M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.
- [2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *Proc. OSDI*. San Francisco, CA, Dec. 2004.
- [3] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Uppsala University, Nov. 2000.
- [4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: Highbandwidth multicast in cooperative environments. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.
- [5] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proc. NSDI*, San Francisco, CA, April 2004.
- [6] F. Cristian and F. Schmuck. Agreeing on processor group membership in timed asynchronous distributed systems. Report CSE95-428, UC San Diego, 1995.
- [7] C. Dickens. Great Expectations. Chapman & Hall, London, 1861.
- [8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. OSDI*, San Diego, CA, Dec. 2000.
- [9] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Proc. APPSEM*, Caminha, Portugal, 2000.
- [10] P. Godefroid. Software model checking: the VeriSoft approach. Formal Methods in System Design, 26(2):77–101, Mar. 2005.
- [11] ISO 9074. Estelle: A formal description technique based on an extended state transition model. 1987.
- [12] Mace. http://mace.ucsd.edu, 2005.
- [13] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX* 2005, Anaheim, CA, Apr. 2005.
- [14] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proc. USITS*, Seattle, WA, Mar. 2003.
- [15] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.
- [16] L. Lamport. The temporal logic of actions. ACM TOPLAS, 16(3):872–923, May 1994.
- [17] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.
- [18] B. T. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative over-

- lays. In Proc. SOSP, Brighton, UK, Oct. 2005.
- [19] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37– 46, Nov. 1995.
- [20] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, Oct. 1991.
- [21] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proc. SOSP*, Asheville, NC, Dec. 1993.
- [22] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. NSDI*, San Francisco, CA, April 2004.
- [23] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. Middleware'2001*, Heidelberg, Germany, Nov. 2001.
- [24] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In 3rd Intl. Workshop on Networked Group Communication, London, UK, Nov. 2001.
- [25] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. ASPLOS*, Boston, MA, 2004.
- [26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI*, Boston, MA, 2002.
- [27] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. SOSP*, Banff, Canada, 2001.

Notes

¹Pip is the main character in *Great Expectations* [7].

²Source code and screenshots for Pip are available at http://issg.cs.duke.edu/pip.

³Mace is an ongoing redesign of the MACEDON [22] language for building distributed systems.

OASIS: Anycast for Any Service

Michael J. Freedman*‡, Karthik Lakshminarayanan†, David Mazières‡
*New York University, †U.C. Berkeley, ‡Stanford University
http://www.coralcdn.org/oasis/

Abstract

Global anycast, an important building block for many distributed services, faces several challenging requirements. First, anycast response must be fast and accurate. Second, the anycast system must minimize probing to reduce the risk of abuse complaints. Third, the system must scale to many services and provide high availability. Finally, and most importantly, such a system must integrate seamlessly with unmodified client applications. In short, when a new client makes an anycast query for a service, the anycast system must ideally return an accurate reply without performing any probing at all.

This paper presents OASIS, a distributed anycast system that addresses these challenges. Since OASIS is shared across many application services, it amortizes deployment and network measurement costs; yet to facilitate sharing, OASIS has to maintain network locality information in an application-independent way. OASIS achieves these goals by mapping different portions of the Internet in advance (based on IP prefixes) to the geographic coordinates of the nearest known landmark. Measurements from a preliminary deployment show that OASIS, surprisingly, provides a significant improvement in the performance that clients experience over state-of-theart on-demand probing and coordinate systems, while incurring much less network overhead.

1 Introduction

Many Internet services are distributed across a collection of servers that handle client requests. For example, high-volume web sites are typically replicated at multiple locations for performance and availability. Content distribution networks amplify a website's capacity by serving clients through a large network of web proxies. File-sharing and VoIP systems use rendezvous servers to bridge hosts behind NATs.

The performance and cost of such systems depend highly on the servers that clients select. For example, file download times can vary greatly based on the locality and load of the chosen replica. Furthermore, a service provider's costs may depend on the load spikes that the server-selection mechanism produces, as many data centers charge customers based on the 95th-percentile usage over all five-minute periods in a month.

Unfortunately, common techniques for replica selection produce sub-optimal results. Asking human users to select the best replica is both inconvenient and inaccurate. Round-robin and other primitive DNS techniques spread load, but do little for network locality.

More recently, sophisticated techniques for serverselection have been developed. When a legacy client initiates an anycast request, these techniques typically probe the client from a number of vantage points, and then use this information to find the closest server. While efforts, such as virtual coordinate systems [6, 28] and on-demand probing overlays [40, 46], seek to reduce the probing overhead, the savings in overhead comes at the cost of accuracy of the system.

Nevertheless, significant on-demand probing is still necessary for all these techniques, and this overhead is reincurred by every new deployed service. While on-demand probing potentially offers greater accuracy, it has several drawbacks that we have experienced first-hand in a previously deployed system [10]. First, probing adds latency, which can be significant for small web requests. Second, performing several probes to a client often triggers intrusion-detection alerts, resulting in abuse complaints. This mundane problem can pose real operational challenges for a deployed system.

This paper presents OASIS (Overlay-based Anycast Service InfraStructure), a shared locality-aware server selection infrastructure. OASIS is organized as an infrastructure overlay, providing high availability and scalability. OASIS allows a service to register a list of servers, then answers the query, "Which server should the client contact?" Selection is primarily optimized for network locality, but also incorporates liveness and load. OASIS can, for instance, be used by CGI scripts to redirect clients to an appropriate web mirror. It can locate servers for IP anycast proxies [2], or it can select distributed SMTP servers in large email services [26].

To eliminate on-demand probing when clients make anycast requests, OASIS probes clients in the background. One of OASIS's main contributions is a set of

Keyword	Threads	Msgs	Keyword	Threads	Msgs	
abuse	198	888	ICMP	64	308	
attack	98	462	IDS	60	222	
blacklist	32	158	intrusion	14	104	
block	168	898	scan	118	474	
complaint	216	984	trojan	10	56	
flood	4	30	virus	24	82	

Figure 1: Frequency count of keywords in PlanetLab *support-community* archives from 14-Dec-04 through 30-Sep-05, comprising 4682 messages and 1820 threads. Values report number of messages and unique threads containing keyword.

techniques that makes it practical to measure the entire Internet in advance. By leveraging the locality of the IP prefixes [12], OASIS probes only each prefix, not each client; in practice, IP prefixes from BGP dumps are used as a starting point. OASIS delegates measurements to the service replicas themselves, thus amortizing costs (approximately 2–10 GB/week) across multiple services, resulting in an acceptable per-node cost.

To share OASIS across services and to make background probing feasible, OASIS requires *stable network coordinates* for maintaining locality information. Unfortunately, virtual coordinates tend to drift over time. Thus, since OASIS seeks to probe an IP prefix as infrequently as once a week, virtual coordinates would not provide sufficient accuracy. Instead, OASIS stores the geographic coordinates of the replica closest to each prefix it maps.

OASIS is publicly deployed on PlanetLab [34] and has already been adopted by a number of services, including ChunkCast [5], CoralCDN [10], Na Kika [14], OCALA [19], and OpenDHT [37]. Currently, we have implemented a DNS redirector that performs server selection upon hostname lookups, thus supporting a wide range of unmodified client applications. We also provide an HTTP and RPC interface to expose its anycast and locality-estimation functions to OASIS-aware hosts.

Experiments from our deployment have shown rather surprisingly that the accuracy of OASIS is competitive with Meridian [46], currently the best on-demand probing system. In fact, OASIS performs better than all replicaselection schemes we evaluated across a variety of metrics, including resolution and end-to-end download times for simulated web sessions, while incurring much less network overhead.

2 Design

An anycast infrastructure like OASIS faces three main challenges. First, network peculiarities are fundamental to Internet-scale distributed systems. Large latency fluctuations, non-transitive routing [11], and middleboxes such as transparent web proxies, NATs, and firewalls can produce wildly inaccurate network measurements and hence suboptimal anycast results.

Second, the system must balance the goals of accuracy, response time, scalability, and availability. In general, using more measurements from a wider range of vantage points should result in greater accuracy. However, probing clients on-demand increases latency and may overemphasize transient network conditions. A better approach is to probe networks in advance. However, services do not know which clients to probe apriori, so this approach effectively requires measuring the whole Internet, a seemingly daunting task.

A shared infrastructure, however, can spread measurement costs over many hosts and gain more network vantage points. Of course, these hosts may not be reliable. While structured peer-to-peer systems [39, 42] can, theoretically, deal well with unreliable hosts, such protocols add significant complexity and latency to a system and break compatibility with existing clients. For example, DNS resolvers and web browsers deal poorly with unavailable hosts since hosts cache stale addresses longer than appropriate.

Third, even with a large pool of hosts over which to amortize measurement costs, it is important to minimize the rate at which any network is probed. Past experience [10] has shown us that repeatedly sending unusual packets to a given destination often triggers intrusion detection systems and results in abuse complaints. For example, PlanetLab's *support-community* mailing list receives thousands of complaints yearly due to systems that perform active probing; Figure 1 lists the number and types of complaints received over one ten-month period. They range from benign inquiries to blustery threats to drastic measures such as blacklisting IP addresses and entire netblocks. Such measures are not just an annoyance; they impair the system's ability to function.

This section describes how OASIS's design tackles the above challenges. A two-tier architecture (§2.1) combines a reliable core of hosts that implement anycast with a larger number of replicas belonging to different services that also assist in network measurement. OASIS minimizes probing and reduces susceptibility to network peculiarities by exploiting geographic coordinates as a basis for locality (§2.2.2). Every replica knows its latitude and longitude, which already provides some information about locality before any network measurement. Then, in the background, OASIS estimates the geographic coordinates of every netblock on the Internet. Because the physical location of IP prefixes rarely changes [36], an accurately pinpointed network can be safely re-probed very infrequently (say, once a week). Such infrequent, background probing both reduces the risk of abuse complaints and allows fast replies to anycast requests with no need for on-demand probing.

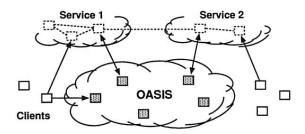


Figure 2: OASIS system overview

2.1 System overview

Figure 2 shows OASIS's high-level architecture. The system consists of a network of *core* nodes that help *clients* select appropriate *replicas* of various services. All services employ the same core nodes; we intend this set of infrastructure nodes to be small enough and sufficiently reliable so that every core node can know most of the others. Replicas also run OASIS-specific code, both to report their own load and liveness information to the core, and to assist the core with network measurements. Clients need not run any special code to use OASIS, because the core nodes provide DNS- and HTTP-based redirection services. An RPC interface is also available to OASIS-aware clients.

Though the three roles of core node, client, and replica are distinct, the same physical host often plays multiple roles. In particular, core nodes are all replicas of the OASIS RPC service, and often of the DNS and HTTP redirection services as well. Thus, replicas and clients typically use OASIS itself to find a nearby core node.

Figure 3 shows various ways in which clients and services can use OASIS. The top diagram shows an OASIS-aware client, which uses DNS-redirection to select a nearby replica of the OASIS RPC service (*i.e.*, a core node), then queries that node to determine the best replica of Service 1.

The middle diagram shows how to make legacy clients select replicas using DNS redirection. The service provider advertises a domain name served by OASIS. When a client looks up that domain name, OASIS first redirects the client's resolver to a nearby replica of the DNS service (which the resolver will cache for future accesses). The nearby DNS server then returns the address of a Service 2 replica suitable for the client. This result can be accurate if clients are near their resolvers, which is often the case [24].

The bottom diagram shows a third technique, based on service-level (e.g., HTTP) redirection. Here the replicas of Service 3 are also clients of the OASIS RPC service. Each replica connects to a nearby OASIS core node selected by DNS redirection. When a client connects to a replica, that replica queries OASIS to find a better replica,

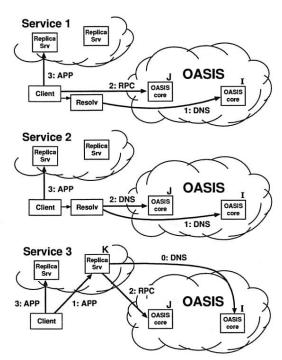


Figure 3: Various methods of using OASIS via its DNS or RPC interfaces, and the steps involved in each anycast request.

then redirects the client. Such an approach does not require that clients be located near their resolvers in order to achieve high accuracy.

This paper largely focuses on DNS redirection, since it is the easiest to integrate with existing applications.

2.2 Design decisions

Given a client IP address and service name, the primary function of the OASIS core is to return a suitable service replica. For example, an OASIS nameserver calls its core node with the client resolver's IP address and a service name extracted from the requested domain name (e.g., coralcdn.nyuld.net indicates service coralcdn).

Figure 4 shows how OASIS resolves an anycast request. First, a core node maps the client IP address to a *network bucket*, which aggregates adjacent IP addresses into netblocks of co-located hosts. It then attempts to map the bucket to a *location* (i.e., coordinates). If successful, OASIS returns the closest service replica to that location (unless load-balancing requires otherwise, as described in §3.4). Otherwise, if it cannot determine the client's location, it returns a random replica.

The anycast process relies on four databases maintained in a distributed manner by the core: (1) a *service table* lists all services using OASIS (and records policy information for each service), (2) a *bucketing table* maps IP addresses to buckets, (3) a *proximity table* maps buckets to locations, and (4) one *liveness table per service* in-

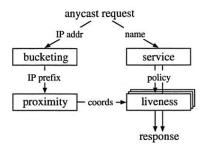


Figure 4: Logical steps to answer an anycast request

cludes all live replicas belonging to the service and their corresponding information (e.g., coordinates, load, and capacity).

2.2.1 Buckets: The granularity of mapping hosts

OASIS must balance the precision of identifying a client's network location with its state requirements. One strawman solution is simply to probe every IP address ever seen and cache results for future requests. Many services have too large a client population for such an approach to be attractive. For DNS redirection, probing each DNS resolver would be practical if the total number of resolvers were small and constant. Unfortunately, measurements at DNS root servers [23] have shown many resolvers use dynamically-assigned addresses, thus precluding a small working set.

Fortunately, our previous research has shown that IP aggregation by prefix often preserves locality [12]. For example, more than 99% of /24 IP prefixes announced by stub autonomous systems (and 97% of /24 prefixes announced by all autonomous systems) are at the same location. Thus, we aggregate IP addresses using IP prefixes as advertised by BGP, using BGP dumps from Route-Views [38] as a starting point. 1

However, some IP prefixes (especially larger prefixes) do not preserve locality [12]. OASIS discovers and adapts to these cases by splitting prefixes that exhibit poor locality precision,² an idea originally proposed by IP2Geo [30]. Using IP prefixes as network buckets not only improves scalability by reducing probing and state requirements, but also provides a concrete set of targets to precompute, and hence avoid on-demand probing.

2.2.2 Geographic coordinates for location

OASIS takes a two-pronged approach to locate IP prefixes: We first use a direct probing mechanism [46] to

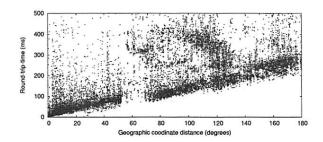


Figure 5: Correlation between round-trip-times and geographic distance across all PlanetLab hosts [43].

find the replica closest to the prefix, regardless of service. Then, we represent the prefix by the geographic coordinates of this closest replica and its measured roundtrip-time to the prefix. We assume that all replicas know their latitude and longitude, which can easily be obtained from a variety of online services [13]. Note that OASIS's shared infrastructure design helps increase the number of vantage points and thus improves its likelihood of having a replica near the prefix.

While geographic coordinates are certainly not optimal predictors of round-trip-times, they work well in practice: The heavy band in Figure 5 shows a strong linear correlation between geographic distance and RTT. In fact, anycast only has the weaker requirement of predicting a relative ordering of nodes for a prefix, not an accurate RTT estimation. For comparison, we also implemented Vivaldi [6] and GNP [28] coordinates within OASIS; §5 includes some comparison results.

Time- and service-invariant coordinates. Since geographic coordinates are stable over time, they allow OA-SIS to probe each prefix infrequently. Since geographic coordinates are independent of the services, they can be shared across services—an important requirement since OASIS is designed as a shared infrastructure. Geographic coordinates remain valid even if the closest replica fails. In contrast, virtual coordinate systems [6, 28] fall short of providing either accuracy or stability [40, 46]. Similarly, simply recording a prefix's nearest replica-without its corresponding geographic coordinates—is useless if that nearest replica fails. Such an approach also requires a separate mapping per service.

Absolute error predictor. Another advantage of our two-pronged approach is that the RTT between a prefix and its closest replica is an absolute bound on the accuracy of the prefix's estimated location. This bound suggests a useful heuristic for deciding when to re-probe a prefix to find a better replica. If the RTT is small (a few milliseconds), reprobing is likely to have little effect. Conversely, reprobing prefixes having high RTTs to their closest replica can help improve accuracy when

¹For completeness, we also note that OASIS currently supports aggregating by the less-locality-preserving autonomous system number, although we do not present the corresponding results in this paper.

²We deem that a prefix exhibits poor locality if probing different IP addresses within the prefix yields coordinates with high variance.

previous attempts missed the best replica or newly-joined replicas are closer to the prefix. Furthermore, a prefix's geographic coordinates will not change unless it is probed by a closer replica. Of course, IP prefixes can physically move, but this happens rarely enough [36] that OASIS only expires coordinates after one week. Moving a network can therefore result in sub-optimal predictions for at most one week.

Sanity checking. A number of network peculiarities can cause incorrect network measurements. For example, a replica behind a transparent web proxy may erroneously measure a short RTT to some IP prefix, when in fact it has only connected to the proxy. Replicas behind firewalls may believe they are pinging a remote network's firewall, when really they are probing their own. OASIS employs a number of tests to detect such situations (see §6). As a final safeguard, however, the core only accepts a prefix-to-coordinate mapping after seeing two consistent measurements from replicas on different networks.

In hindsight, another benefit of geographic coordinates is the ability to couple them with real-time visualization of the network [29], which has helped us identify, debug, and subsequently handle various network peculiarities.

2.2.3 System management and data replication

To achieve scalability and robustness, the location information of prefixes must be made available to all core nodes. We now describe OASIS's main system management and data organization techniques.

Global membership view. Every OASIS core node maintains a weakly-consistent view of all other nodes in the core, where each node is identified by its IP address, a globally-unique node identifier, and an incarnation number. To avoid $O(n^2)$ probing (where n is the network size), core nodes detect and share failure information cooperatively: every core node probes a random neighbor each time period (3 seconds) and, if it fails to receive a response, gossips its suspicion of failure.

Two techniques suggested by SWIM [7] reduce false failure announcements. First, several intermediates are chosen to probe this target before the initiator announces its suspicion of failure. Intermediaries alleviate the problems caused by non-transitive Internet routing [11]. Second, incarnation numbers help disambiguate failure messages: alive messages for incarnation i override anything for j < i; suspect for i overrides anything for $j \le i$. If a node learns that it is suspected of failure, it increments its incarnation number and gossips its new number as alive. A node will only conclude that another node with incarnation i is dead if it has not received a corresponding alive message for j > i after some time (3 minutes). This ap-

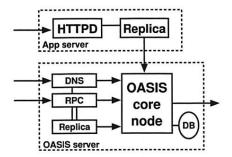


Figure 6: OASIS system components

proach provides live nodes with sufficient time to respond to and correct false suspicions of failure.

Implicit in this design is the assumption that nodes are relatively stable; otherwise, the system would incur a high bandwidth cost for failure announcements. Given that OASIS is designed as an *infrastructure service*—to be deployed either by one service provider or a small number of cooperating providers—we believe that this assumption is reasonable.

Consistent hashing. OASIS tasks must be assigned to nodes in some globally-known yet fully-decentralized manner. For example, to decide the responsibility of mapping specific IP prefixes, we partition the set of prefixes over all nodes. Similarly, we assign specific nodes to play the role of a *service rendezvous* to aggregate information about a particular service (described in §3.3).

OASIS provides this assignment through consistent hashing [20]. Each node has a random identifier; several nodes with identifiers closest to a key—e.g., the SHA-1 hash of the IP prefix or service name—in the identifier space are assigned the corresponding task. Finding these nodes is easy since all nodes have a global view. While nodes' views of the set of closest nodes are not guaranteed to be consistent, views can be easily reconciled using nodes' incarnation numbers.

Gossiping. OASIS uses gossiping to efficiently disseminate messages—about node failures, service policies, prefix coordinates—throughout the network [7]. Each node maintains a buffer of messages to be piggybacked on other system messages to random nodes. Each node gossips each message $O(\log n)$ times for n-node networks; such an epidemic algorithm propagates a message to all nodes in logarithmic time with high probability.³

Soft-state replica registration. OASIS must know all replicas belonging to a service in order to answer corresponding anycast requests. To tolerate replica failures robustly, replica information is maintained using soft-state:

³While structured gossiping based on consistent hashing could reduce the bandwidth overhead needed to disseminate a message [3], we use a randomized epidemic scheme for simplicity.

replicas periodically send registration messages to core nodes (currently, every 60 seconds).

Hosts running services that use OASIS for anycast—such as the web server shown in Figure 6—run a separate replica process that connects to their local application (i.e., the web server) every keepalive period (currently set to 15 seconds). The application responds with its current load and capacity. While the local application remains alive, the replica continues to refresh its locality, load, and capacity with its OASIS core node.

Closest-node discovery. OASIS offloads all measurement costs to service replicas. All replicas, belonging to different services, form a lightweight overlay, in order to answer closest-replica queries from core nodes. Each replica organizes its neighbors into concentric rings of exponentially-increasing radii, as proposed by Meridian [46]: A replica accepts a neighbor for ring i only if its RTT is between 2^i and 2^{i+1} milliseconds. To find the closest replica to a destination d, a query operates in successive steps that "zero in" on the closest node in an expected $O(\log n)$ steps. At each step, a replica with RTT r from d chooses neighbors to probe d, restricting its selection to those with RTTs (to itself) between $\frac{1}{2}r$ and $\frac{3}{2}r$. The replica continues the search on its neighbor returning the minimum RTT to d. The search stops when the latest replica knows of no other potentially-closer nodes.

Our implementation differs from [46] in that we perform closest routing iteratively, as opposed to recursively: The first replica in a query initiates each progressive search step. This design trades overlay routing speed for greater robustness to packet loss.

3 Architecture

In this section, we describe the distributed architecture of OASIS in more detail: its distributed management and collection of data, locality and load optimizations, scalability, and security properties.

3.1 Managing information

We now describe how OASIS manages the four tables described in §2.2. OASIS optimizes response time by heavily replicating most information. Service, bucketing, and proximity information need only be weakly consistent; stale information only affects system performance, not its correctness. On the other hand, replica liveness information must be more fresh.

Service table. When a service initially registers with OASIS, it includes a service policy that specifies its service name and any domain name aliases, its desired server-selection algorithm, a public signature key, the

maximum and minimum number of addresses to be included in responses, and the TTLs of these responses. Each core node maintains a local copy of the service table to be able to efficiently handle requests. When a new service joins OASIS or updates its existing policy, its policy is disseminated throughout the system by gossiping.

The server-selection algorithm specifies how to order replicas as a function of their distance, load, and total capacity when answering anycast requests. By default, OASIS ranks nodes by their coordinate distance to the target, favoring nodes with excess capacity to break ties. The optional signature key is used to authorize replicas registering with an OASIS core node as belonging to the service (see §3.5).

Bucketing table. An OASIS core node uses its bucketing table to map IP addresses to IP prefixes. We bootstrap the table using BGP feeds from RouteViews [38], which has approximately 200,000 prefixes. A PATRICIA trie [27] efficiently maps IP addresses to prefixes using longest-prefix matching.

When core nodes modify their bucketing table by splitting or merging prefixes [30], these changes are gossiped in order to keep nodes' tables weakly consistent. Again, stale information does not affect system correctness: prefix withdrawals are only used to reduce system state, while announcements are used only to identify more precise coordinates for a prefix.

Proximity table. When populating the proximity table, OASIS seeks to find accurate coordinates for every IP prefix, while preventing unnecessary reprobing.

OASIS maps an IP prefix to the coordinates of its closest replica. To discover the closest replica, an core node first selects an IP address from within the prefix and issues a probing request to a known replica (or first queries a neighbor to discover one). The selected replica traceroutes the requested IP to find the last routable IP address, performs closest-node discovery using the replica overlay (see §2.2.3), and, finally, returns the coordinates of the nearest replica and its RTT distance from the target IP. If the prefix's previously recorded coordinate has either expired or has a larger RTT from the prefix, the OASIS core node reassigns the prefix to these new coordinates and starts gossiping this information.

To prevent many nodes from probing the same IP prefix, the system assigns prefixes to nodes using consistent hashing. That is, several nodes closest to hash(prefix) are responsible for probing the prefix (three by default). All nodes go through their subset of assigned prefixes in random order, probing the prefix if its coordinates have not been updated within the last T_p seconds. T_p is a function of the coordinate's error, such that highly-accurate coordinates are probed at a slower rate (see §2.2.2).

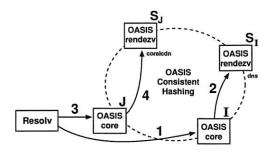


Figure 7: Steps involved in a DNS anycast request to OASIS using rendezvous nodes.

Liveness table. For each registered service, OASIS maintains a liveness table of known replicas. Gossiping is not appropriate to maintain these liveness tables at each node: stale information could cause nodes to return addresses of failed replicas, while high replica churn would require excessive gossiping and hence bandwidth consumption.

Instead, OASIS aggregates liveness information about a particular service at a few *service rendezvous* nodes, which are selected by consistent hashing. When a replica joins or leaves the system, or undergoes a significant load change, the OASIS core node with which it has registered sends an update to one of the k nodes closest to hash(service). For scalability, these rendezvous nodes only receive occasional state updates, not each soft-state refresh continually sent by replicas to their core nodes. Rendezvous nodes can dynamically adapt the parameter k based on load, which is then gossiped as part of the service's policy. By default, k=4, which is also fixed as a lower bound.

Rendezvous nodes regularly exchange liveness information with one another, to ensure that their liveness tables remain weakly consistent. If a rendezvous node detects that an core node fails (via OASIS's failure detection mechanism), it invalidates all replicas registered by that node. These replicas will subsequently re-register with a different core node and their information will be re-populated at the rendezvous nodes.

Compared to logically-decentralized systems such as DHTs [39, 42], this aggregation at rendezvous nodes allows OASIS to provide faster response (similar to one-hop lookups) and to support complex anycast queries (e.g., as a function of both locality and load).

3.2 Putting it together: Resolving anycast

Given the architecture that we have presented, we now describe the steps involved when resolving an anycast request (see Figure 7). For simplicity, we limit our discussion to DNS redirection. When a client queries OASIS for the hostname *coralcdn.nyuld.net* for the first time:

- 1. The client queries the DNS root servers, finding an OASIS nameserver *I* for *nyuld.net* to which it sends the request.
- Core lookup: OASIS core node I finds other core nodes near the client that support the DNS interface by executing the following steps:
 - (a) *I* locally maps the client's IP address to IP prefix, and then prefix to location coordinates.
 - (b) I queries one of the k rendezvous nodes for service dns, call this node S_I, sending the client's coordinates.
 - (c) S_I responds with the best-suited OASIS name-servers for the specified coordinates.
 - (d) *I* returns this set of DNS replicas to the client. Let this set include node *J*.
- 3. The client resends the anycast request to J.
- 4. **Replica lookup:** Core node *J* finds replicas near the client using the following steps:
 - (a) J extracts the request's service name and maps the client's IP address to coordinates.
 - (b) J queries one of the k rendezvous nodes for service *coralcdn*, call this S_J .
 - (c) S_J responds with the best *coralcdn* replicas, which J returns to the client.

Although DNS is a stateless protocol, we can force legacy clients to perform such two-stage lookups, as well as signal to their nameservers which stage they are currently executing. §4 gives implementation details.

3.3 Improving scalability and latency

While OASIS can support a large number of replicas by simply adding more nodes, the anycast protocol described in $\S 3.2$ has a bottleneck in scaling to large numbers of clients for a particular service: one of the k rendezvous nodes is involved in each request. We now describe how OASIS reduces these remote queries to improve both scalability and client latency.

Improving core lookups. OASIS first reduces load on rendezvous nodes by lowering the frequency of core lookups. For DNS-based requests, OASIS uses relatively-long TTLs for OASIS nameservers (currently 15 minutes) compared to those for third-party replicas (configurable per service, 60 seconds by default). These longer TTLs seem acceptable given that OASIS is an infrastructure service, and that resolvers can failover between nameservers since OASIS returns multiple, geodiverse nameservers.

Second, we observe that core lookups are rarely issued to *random* nodes: Core lookups in DNS will initially go to one of the twelve primary nameservers registered for *.nyuld.net* in the main DNS hierarchy. So, we can arrange the OASIS core so that these 12 primary nameservers play the role of rendezvous nodes for dns, by simply having them choose k=12 consecutive node identifiers for consistent hashing (in addition to their normal random identifiers). This configuration reduces latency by avoiding remote lookups.

Improving replica lookups. OASIS further reduces load by leveraging request locality. Since both clients and replicas are redirected to their nearest OASIS core nodes—when performing anycast requests and initiating registration, respectively—hosts redirected to the same core node are likely to be close to one another. Hence, on receiving a replica lookup, an core node first checks its local liveness table for any replica that satisfies the service request.

To improve the effectiveness of using local information, OASIS also uses *local flooding*: Each core node receiving registrations sends these local replica registrations to some of its closest neighbors. ("Closeness" is again calculated using coordinate distance, to mirror the same selection criterion used for anycast.) Intuitively, this approach helps prevent situations in which replicas and clients select different co-located nodes and therefore lose the benefit of local information. We analyze the performance benefit of local flooding in §5.1.

OASIS implements other obvious strategies to reduce load, including having core nodes cache replica information returned by rendezvous nodes and batch replica updates to rendezvous nodes. We do not discuss these further due to space limitations.

3.4 Selecting replicas based on load

While our discussion has mostly focused on locality-based replica selection, OASIS supports multiple selection algorithms incorporating factors such as load and capacity. However, in most practical cases, load-balancing need not be perfect; a reasonably good node is often acceptable. For example, to reduce costs associated with "95th-percentile billing," only the elimination of traffic spikes is critical. To eliminate such spikes, a service's replicas can track their 95% bandwidth usage over five-minute windows, then report their load to OASIS as the logarithm of this bandwidth usage. By specifying load-based selection in its policy, a service can ensure that its 95% bandwidth usage at its most-loaded replica is within a factor of two of its least-loaded replica; we have evaluated this policy in §5.2.

However, purely load-based metrics cannot be used in conjunction with many of the optimizations that reduce replica lookups to rendezvous nodes (§3.3), as locality does not play a role in such replica selection. On the

other hand, the computation performed by rendezvous nodes when responding to such replica lookups is much lower: while answering locality-based lookups requires the rendezvous node to compute the closest replica(s) with respect to the client's location, answering load-based lookups requires the node simply to return the first element(s) of a single list of service replicas, sorted by increasing load. The ordering of this list needs to be recomputed only when replicas' loads change.

3.5 Security properties

OASIS has the following security requirements. First, it should prohibit unauthorized replicas from joining a registered service. Second, it should limit the extent to which a particular service's replicas can inject bad coordinates. Finally, it should prevent adversaries from using the infrastructure as a platform for DDoS attacks.

We assume that all OASIS core nodes are trusted; they do not gossip false bucketing, coordinates, or liveness information. We also assume that core nodes have loosely synchronized clocks to verify expiry times for replicas' authorization certificates. (Loosely-synchronized clocks are also required to compare registration expiry times in liveness tables, as well as measurement times when determining whether to reprobe prefixes.) Additionally, we assume that services joining OASIS have some secure method to initially register a public key. An infrastructure deployment of OASIS may have a single or small number of entities performing such admission control; the service provider(s) deploying OASIS's primary DNS nameservers are an obvious choice. Less secure schemes such as using DNS TXT records may also be appropriate in certain contexts.

To prevent unauthorized replicas from joining a service, a replica must present a valid, fresh certificate signed by the service's public key when initially registering with the system. This certificate includes the replica's IP address and its coordinates. By providing such admission control, OASIS only returns IP addresses that are authorized as valid replicas for a particular service.

OASIS limits the extent to which replicas can inject bad coordinates by evicting faulty replicas or their corresponding services. We believe that sanity-checking coordinates returned by the replicas—coupled with the penalty of eviction—is sufficient to deter services from assigning inaccurate coordinates for their replicas and replicas from responding falsely to closest-replica queries from OASIS.

Finally, OASIS prevents adversaries from using it as a platform for distributed denial-of-service attacks by requiring that replicas accept closest-replica requests only from core nodes. It also requires that a replica's overlay neighbors are authorized by OASIS (hence, replicas

Figure 8: Output of dig for a hostname using OASIS.

only accept probing requests from other approved replicas). OASIS itself has good resistance to DoS attacks, as most client requests can be resolved using information stored locally, *i.e.*, not requiring wide-area lookups between core nodes.

4 Implementation

OASIS's implementation consists of three main components: the OASIS core node, the service replica, and stand-alone interfaces (including DNS, HTTP, and RPC). All components are implemented in C++ and use the asynchronous I/O library from the SFS toolkit [25], structured using asynchronous events and callbacks. The core node comprises about 12,000 lines of code, the replica about 4,000 lines, and the various interfaces about 5,000 lines. The bucketing table is maintained using an inmemory PATRICIA trie [27], while the proximity table uses BerkeleyDB [41] for persistent storage.

OASIS's design uses static latitude/longitude coordinates with Meridian overlay probing [46]. For comparison purposes, OASIS also can be configured to use synthetic coordinates using Vivaldi [6] or GNP [28].

RPC and HTTP interfaces. These interfaces take an optional target IP address as input, as opposed to simply using the client's address, in order to support integration of third-party services such as HTTP redirectors (Figure 3). Beyond satisfying normal anycast requests, these interfaces also enable a localization service by simply exposing OASIS's proximity table, so that any client can ask "What are the coordinates of IP x?" In addition to HTML, the HTTP interface supports XML-formatted output for easy visualization using online mapping services [13].

DNS interface. OASIS takes advantage of low-level DNS details to implement anycast. First, a nameserver must differentiate between core and replica lookups. Core lookups only return *nameserver* (NS) records for

nearby OASIS nameservers. Replica lookups, on the other hand, return *address* (A) records for nearby replicas. Since DNS is a stateless protocol, we signal the type of a client's request in its DNS query: replica lookups all have *oasis* prepended to *nyuld.net*. We force such signalling by returning CNAME records during core lookups, which map aliases to their *canonical names*.

This technique alone is insufficient to force many client resolvers, including BIND, to immediately issue replica lookups to these nearby nameservers. We illustrate this with an example query for CoralCDN [10], which uses the service alias *.nyud.net. A resolver R discovers nameservers u, v for nyud.net by querying the root servers for example.net.nyud.net. Next, R queries u for this hostname, and is returned a CNAME for example.net.nyud.net \rightarrow coralcdn.oasis.nyuld.net and NS x, y for coralcdn.oasis.nyuld.net. In practice, R will reissue a new query for coralcdn.oasis.nyuld.net to nameserver v, which is not guaranteed to be close to R (and v's local cache may include replicas far from R).

We again use the DNS query string to signal whether a client is contacting the correct nameservers. When responding to core lookups, we encode the set of NS records in hex format (ab4040d9a9e53205) in the returned CNAME record (Figure 8). Thus, when v receives a replica lookup, it checks whether the query encodes its own IP address, and if it does not, immediately re-returns NS records for x,y. Now, having received NS records authoritative for the name queried, a resolver contacts the desired nameservers x or y, which returns an appropriate replica for coralcdn.

5 Evaluation

We evaluate OASIS's performance benefits for DNSbased anycast, as well as its scalability and bandwidth trade-offs.

5.1 Wide-area evaluation of OASIS

Experimental setup. We present wide-area measurements on PlanetLab [34] that evaluate the accuracy of replica selection based on round-trip-time and throughput, DNS response time, and the end-to-end time for a simulated web session. In all experiments, we ran replicas for one service on approximately 250 PlanetLab hosts spread around the world (including 22 in Asia), and we ran core nodes and DNS servers on 37 hosts.⁶

⁴We plan to support such functionality with DNS TXT records as well, although this has not been implemented yet.

⁵To adopt OASIS yet preserve its own top-level domain name, CoralCDN points the NS records for *nyud.net* to OASIS's nameservers; *nyud.net* is registered as an alias for *coralcdn* in its service policy.

⁶This number was due to the unavailability of UDP port 53 on most PlanetLab hosts, especially given CoralCDN's current use of same.

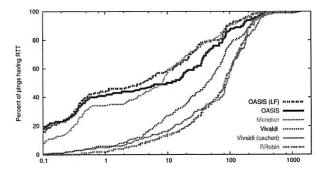


Figure 9: Round trip times (ms)

We compare the performance of replica selection using six different anycast strategies: (1) OASIS (LF) refers to the OASIS system, using both local caching and local flooding (to the nearest three neighbors; see §3.3). (2) OASIS uses only local caching for replicas. (3) Meridian (our implementation of [46]) performs on-demand probing by executing closest-replica discovery whenever it receives a request. (4) Vivaldi uses 2-dimensional dynamic virtual coordinates [6], instead of static geographic coordinates, by probing the client from 8-12 replicas on-demand. The core node subsequently computes the client's virtual coordinates and selects its closest replica based on virtual coordinate distance. (5) Vivaldi (cached) probes IP prefixes in the background, instead of ondemand. Thus, it is similar to OASIS with local caching, except for using virtual coordinates to populate OASIS's proximity table. (6) Finally, RRobin performs roundrobin DNS redirection amongst all replicas in the system, using a single DNS server located at Stanford University.

We performed client measurements on the same hosts running replicas. However, we configured OASIS so that when a replica registers with an OASIS core node, the node does *not* directly save a mapping from the replica's prefix to its coordinates, as OASIS would do normally. Instead, we rely purely on OASIS's background probing to assign coordinates to the replica's prefix.

Three consecutive experiments were run at each site when evaluating ping, DNS, and end-to-end latencies. Short DNS TTLs were chosen to ensure that clients contacted OASIS for each request. Data from all three experiments are included in the following cumulative distribution function (CDF) graphs.

Minimizing RTTs. Figures 9 shows the CDFs of round-trip-times in log-scale between clients and their returned replicas. We measured RTTs via ICMP echo messages, using the ICMP response's kernel timestamp when calculating RTTs. RTTs as reported are the minimum of ten consecutive probes. We see that OASIS and Meridian significantly outperform anycast using Vivaldi and round robin by one to two orders of magnitude.

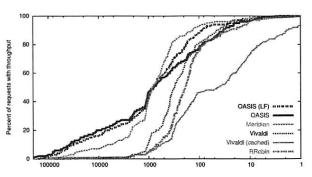


Figure 10: Client-server TCP throughput (KB/s)

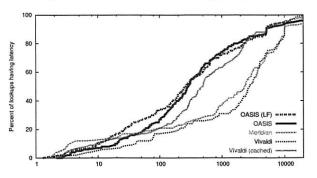


Figure 11: DNS resolution time (ms) for new clients

Two other interesting results merit mention. First, Vivaldi (cached) performs significantly worse than ondemand Vivaldi and even often worse than RRobin. This arises from the fact that Vivaldi is not stable over time with respect to coordinate translation and rotation. Hence, cached results quickly become inaccurate, although recent work has sought to minimize this instability [8, 33]. Second, OASIS outperforms Meridian for 60% of measurements, a rather surprising result given that OASIS uses Meridian as its background probing mechanism. It is here where we see OASIS's benefit from using RTT as an absolute error predictor for coordinates (§2.2.2): reprobing by OASIS yields strictly better results, while the accuracy of Meridian queries can vary.

Maximizing throughput. Figure 10 shows the CDFs of the steady-state throughput from replicas to their clients, to examine the benefit of using nearby servers to improve data-transfer rates. TCP throughput is measured using iperf-1.7.0 [18] in its default configuration (a TCP window size of 32 KB). The graph shows TCP performance in steady-state. OASIS is competitive with or superior to all other tested systems, demonstrating its performance for large data transfers.

DNS resolution time. Figures 11 and 12 evaluate the DNS performance for new clients and for clients already caching their nearby OASIS nameservers, respectively. A request by a new client includes the time to

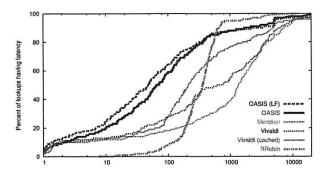


Figure 12: DNS resolution time (ms) for replica lookups

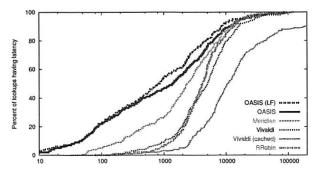


Figure 13: End-to-end download performance (ms)

perform three steps: (1) contact an initial OASIS core node to learn a nearby nameserver, (2) re-contact a distant node and again receive NS records for the same nearby nameservers (see §4), and (3) contact a nearby core node as part of a replica lookup. Note that we did not specially configure the 12 primary nameservers as rendezvous nodes for *dns* (see §3.3), and thus use a widearea lookup during Step 1. This two-step approach is taken by all systems: *Meridian* and *Vivaldi* both perform on-demand probing twice. We omit *RRobin* from this experiment, however, as it always uses a single nameserver. Clients already caching nameserver information need only perform Step 3, as given in Figure 12.

OASIS's strategy of first finding nearby nameservers and then using locally-cached information can achieve significantly faster DNS response times compared to ondemand probing systems. The median DNS resolution time for OASIS replica lookups is almost 30x faster than that for *Meridian*. We also see that local flooding can improve median performance by 40% by reducing the number of wide-area requests to rendezvous nodes.

End-to-end latency. Figure 13 shows the end-to-end time for a client to perform a synthetic web session, which includes first issuing a replica lookup via DNS and then downloading eight 10KB files sequentially. This

metric	california	texas	new york	germany
latency	23.3	0.0	0.0	0.0
load	9.0	11.3	9.6	9.2

Table 1: 95th-percentile bandwidth usage (MB)

file size is chosen to mimic that of common image files, which are often embedded multiple times on a given web page. We do not simulate persistent connections for our transfers, so each request establishes a new TCP connection before downloading the file. Also, our fauxwebserver never touches the disk, so does not take (PlanetLab's high) disk-scheduling latency into account.

End-to-end measurements underscore OASIS's true performance benefit, coupling very fast DNS response time with very accurate server selection. Median response-time for OASIS is 290% faster than Meridian and 500% faster than simple round-robin systems.

5.2 Load-based replica selection

This section considers replica selection based on load. We do not seek to quantify an optimal load- and latency-aware selection metric; rather, we verify OASIS's ability to perform load-aware anycast. Specifically, we evaluate a load-balancing strategy meant to reduce costs associated with 95th-percentile billing (§3.4).

In this experiment, we use four distributed servers that run our faux-webserver. Each webserver tracks its bandwidth usage per minute, and registers its load with its local replica as the logarithm of its 95th-percentile usage. Eight clients, all located in California, each make 50 anycast requests for a 1MB file, with a 20-second delay between requests. (DNS records have a TTL of 15 seconds.)

Table 1 shows that the webserver with highest bandwidth costs is easily within a factor of two of the least-loaded server. On the other hand, locality-based replica selection creates a traffic spike at a single webserver.

5.3 Scalability

Since OASIS is designed as an infrastructure system, we now verify that a reasonable-sized OASIS core can handle Internet-scale usage.

Measurements at DNS root servers have shown steady traffic rates of around 6.5M A queries per 10 minute interval across all $\{e,i,k,m\}$. root-servers.net [23]. With a deployment of 1000 OASIS DNS servers—and, for simplicity, assuming an even distribution of requests to nodes—even if OASIS received requests at an equivalent rate, each node would see only 10 requests per second.

On the other hand, OASIS often uses shorter TTLs to handle replica failover and load balancing. The same datasets showed approximately 100K unique resolvers

⁷A recursive Meridian implementation [46] may be faster than our iterative implementation: our design emphasizes greater robustness to packet loss, given our preference for minimizing probing.

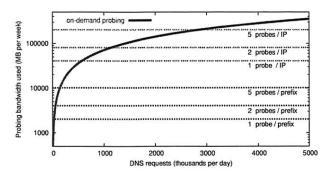


Figure 14: Bandwidth trade-off between on-demand probing, caching IP prefixes (OASIS), and caching IP addresses

per 10 minute interval. Using the default TTL of 60 seconds, even if every client re-issued a request every 60 seconds for all s services using OASIS, each core node would receive at most $1.6 \cdot s$ requests per second.

To consider one real-world service, as opposed to some upper bound for all Internet traffic, CoralCDN [10] handles about 20 million HTTP requests from more than one million unique client IPs per day (as of December 2005). To serve this web population, CoralCDN answers slightly fewer than 5 million DNS queries (for all query types) per day, using TTLs of 30-60 seconds. This translates to a system *total* of 57 DNS queries per second.

5.4 Bandwidth trade-offs

This section examines the bandwidth trade-off between precomputing prefix locality and performing on-demand probing. If a system receives only a few hundred requests per week, OASIS's approach of probing every IP prefix is not worthwhile. Figure 14 plots the amount of bandwidth used in caching and on-demand anycast systems for a system with 2000 replicas. Following the results of [46], we estimate each closest-replica query to generate about 10.4 KB of network traffic (load grows sub-linearly with the number of replicas).

Figure 14 simulates the amount of bandwidth used per week for up to 5 million DNS requests per day (the request rate from CoralCDN), where each results in a new closest-replica query. OASIS's probing of 200K prefixes—even when each prefix may be probed multiple times—generates orders of magnitude less network traffic. We also plot an upper-bound on the amount of traffic generated if the system were to cache IP addresses, as opposed to IP prefixes.

While one might expect the number of DNS resolvers to be constant and relatively small, many resolvers use dynamically-assigned addresses and thus preclude a small working set: the root-servers saw more than 4 mil-

Project	Service	Description
ChunkCast [5]	chunkcast	Anycast gateways
CoralCDN [10]	coralcdn	Web proxies
Na Kika [14]	nakika	Web proxies
OASIS	dns	DNS interface
	http	HTTP interface
	rpc	RPC interface
OCALA [19]	ocala	Client IP gateways
92 889	ocalarsp	Server IP gateways
OpenDHT [37]	opendht	Client DHT gateways

Figure 15: Services using OASIS as of March 2006. Services can be accessed using (service).nyuld.net.

lion unique clients in a week, with the number of clients increasing linearly after the first day's window [23]. Figure 14 uses this upper-bound to plot the amount of traffic needed when caching IP addresses. Of course, new IP addresses always need to be probed on-demand, with the corresponding performance hit (per Figure 12).

6 Deployment lessons

OASIS has been deployed on about 250 PlanetLab hosts since November 2005. Figure 15 lists the systems currently using OASIS and a brief description of their service replicas. We present some lessons that we learned in the process.

Make it easy to integrate. Though each application server requires a local replica, for a shared testbed such as PlanetLab, a single replica process on a host can serve on behalf of multiple local processes running different applications. To facilitate this, we now run OASIS replicas as a public service on PlanetLab; to adopt OASIS, PlanetLab applications need only listen on a registered port and respond to keepalive messages.

Applications can integrate OASIS even without any source-code changes or recompilation. Operators can run or modify simple stand-alone scripts we provide that answer replica keepalive requests after simple liveness and load checks (via ps and the /proc filesystem).

Check for proximity discrepancies. Firewalls and middleboxes can lead one to draw false conclusions from measurement results. Consider the following two problems we encountered, mentioned earlier in §2.2.2.

To determine a routable IP address in a prefix, a replica performs a traceroute and uses the last reachable node that responded to the traceroute. However, since firewalls can perform egress filtering on ICMP packets, an unsuspecting node would then ask others to probe its own egress point, which may be far away from the desired prefix. Hence, replicas initially find their immedi-

ate upstream routers—i.e., the set common to multiple traceroutes—which they subsequently ignored.

When replicas probe destinations on TCP port 80 for closest-replica discovery, any local transparent web proxy will perform full TCP termination, leading an unsuspecting node to conclude that it is very close to the destination. Hence, a replica first checks for a transparent proxy, then tries alternative probing techniques.

Both problems would lead replicas to report themselves as incorrectly close to some IP prefix. So, by employing measurement redundancy, OASIS can compare answers for precision and sanity.

Be careful what you probe. No single probing technique is both sufficiently powerful and innocuous (from the point-of-view of intrusion-detection systems). As such, OASIS has adapted its probing strategies based on ISP feedback. ICMP probes and TCP probes to random high ports were often dropped by egress firewalls and, for the latter, flagged as unwanted port scans. Probing to TCP port 80 faced the problem of transparent web proxies, and probes to TCP port 22 were often flagged as SSH login attacks. Unfortunately, as OASIS performs probing from multiple networks, automated abuse complaints from IDSs are sent to many separate network operators. Currently, OASIS uses a mix of TCP port 80 probes, ICMP probes, and reverse DNS name queries.

Be careful whom you probe. IDSs deployed on some networks are incompatible with active probing, irrespective of the frequency of probes. Thus, OASIS maintains and checks a blacklist whenever a target IP prefix or address is selected for probing. We apply this blacklist at all stages of probing: Initially, only the OASIS core checked target IP prefixes. However, this strategy led to abuse complaints from ASes that provide transit for the target, yet filter ICMPs; in such cases, replicas tracerouting the prefix would end up probing the upstream AS.

7 Related work

We classify related work into two areas most relevant to OASIS: network distance estimation and server selection. Network distance estimation techniques are used to identify the location and/or distance between hosts in the network. The server-selection literature deals with finding an appropriately-located server (possibly using network distance estimation) for a client request.

Network distance estimation. Several techniques have been proposed to reduce the amount of probing per request. Some initial proposals (such as [16]) are based on the triangle-inequality assumption. IDMaps [9] proposed deploying *tracers* that all probe one another; the distance between two hosts is calculated as the sum of

the distances between the hosts and their selected tracers, and between the two selected tracers. Theilmann and Rothermel described a hierarchical tree-like system [44], and Iso-bar proposed a two-tier system using landmark-based clustering algorithms [4]. King [15] used recursive queries to remote DNS nameservers to measure the RTT distance between any two *non-participating* hosts.

Recently, virtual coordinate systems (such as GNP [28] and Vivaldi [6]) offer new methods for latency estimation. Here, nodes generate synthetic coordinates after probing one another. The distance between peers in the coordinate space is used to predict their RTT, the accuracy of which depends on how effectively the Internet can be embedded into a *d*-dimensional (usually Euclidean) space.

Another direction for network estimation has been the use of geographic mapping techniques; the main idea is that if geographic distance is a good indicator of network distance, then estimating geographic location accurately would obtain a first approximation for the network distance between hosts. Most approaches in geographic mapping are heuristic. The most common approaches include performing queries against a whois database to extract city information [17, 32], or tracerouting the address space and then mapping router names to locations based on ISP-specific naming conventions [12, 30]. Commercial entities have sought to create exhaustive IPrange mappings [1, 35].

Server selection. IP anycast was proposed as a network-level solution to server selection [22, 31]. However, with various deployment and scalability problems, IP anycast is not widely used or available. Recently, PIAS has argued for supporting IP anycast as a proxy-based service to overcome deployment challenges [2]; OASIS can serve as a powerful and flexible server-selection backend for such a system.

One of the largest deployed content distribution networks, Akamai [1] reportedly traceroutes the IP address space from multiple vantage points to detect route convergence, then pings the common router from every data center hosting an Akamai cluster [4]. OASIS's task is more difficult than that of commercial CDNs, given its goal of providing anycast for multiple services.

Recent literature has proposed techniques to minimize such exhaustive probing. Meridian [46] (used for DNS redirection by [45]) creates an overlay network with neighbors chosen from a particular distribution; routing to closer nodes is guaranteed to find a minimum given a growth-restricted metric space [21]. In contrast, OASIS completely eliminates on-demand probing.

OASIS allows more flexible server selection than pure locality-based solutions, as it stores load and capacity estimates from replicas in addition to locality information.

8 Conclusion

OASIS is a global distributed anycast system that allows legacy clients to find nearby or unloaded replicas for distributed services. Two main features distinguish OASIS from prior systems. First, OASIS allows multiple application services to share the anycast service. Second, OASIS avoids on-demand probing when clients initiate requests. Measurements from a preliminary deployment show that OASIS, provides a significant improvement in the performance that clients experience over state-of-theart on-demand probing and coordinate systems, while incurring much less network overhead.

OASIS's contributions are not merely its individual components, but also the deployed system that is immediately usable by both legacy clients and new services. Publicly deployed on PlanetLab, OASIS has already been adopted by a number of distributed services [5, 10, 14, 19, 37].

Acknowledgments. We thank A. Nicolosi for the keyword analysis of Figure 1. M. Huang, R. Huebsch, and L. Peterson have aided our efforts to run PlanetLab services. We also thank D. Andersen, S. Annapureddy, N. Feamster, J. Li, S. Rhea, I. Stoica, our anonymous reviewers, and our shepherd, S. Gribble, for comments on drafts of this paper. This work was conducted as part of Project IRIS under NSF grant ANI-0225660.

References

- [1] Akamai Technologies. http://www.akamai.com/, 2006.
- [2] H. Ballani and P. Francis. Towards a global IP anycast service. In SIGCOMM, 2005.
- [3] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In NSDI, May 2005.
- [4] Y. Chen, K. H. Lim, R. H. Katz, and C. Overton. On the stability of network distance estimation. SIGMETRICS Perform. Eval. Rev., 30(2):21–30, 2002.
- [5] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In IPTPS, Feb. 2006.
- [6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In SIGCOMM, Aug. 2004.
- [7] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In Dependable Systems and Networks, June 2002.
- [8] C. de Launois, S. Uhlig, and O. Bonaventure. A stable and distributed network coordinate system. Technical report, Universite Catholique de Louvain, Dec. 2004.
- [9] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Trans. on Networking*, Oct. 2001.
- [10] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In NSDI, Mar. 2004.
- [11] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In WORLDS, Dec. 2005.
- [12] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic locality of IP prefixes. In *IMC*, Oct. 2005.
- [13] Google Maps. http://maps.google.com/, 2006.
- [14] R. Grimm, G. Lichtman, N. Michalakis, A. Elliston, A.Kravetz, J. Miller, and S. Raza. Na Kika: Secure service execution and composition in an open edge-side computing network. In NSDI, May 2006.

- [15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *IMW*, 2001.
- [16] J. Guyton and M. Schwartz. Locating nearby copies of replicated Internet servers. In SIGCOMM, Aug. 1995.
- [17] IP to Lat/Long server, 2005. http://cello.cs.uiuc.edu/cgibin/slamm/ip2ll/.
- [18] Iperf. Version 1.7.0 the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, 2005.
- [19] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In NSDI, May 2006.
- [20] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In STOC, May 1997.
- [21] D. R. Karger and M. Ruhl. Finding nearest neighbors in growthrestricted metrics. In STOC, 2002.
- [22] D. Katabi and J. Wrocławski. A framework for scalable global IP-anycast (GIA). In SIGCOMM, Aug. 2000.
- [23] K. Keys. Clients of DNS root servers, 2002-08-14. http://www.caida.org/projects/dns-analysis/, 2002.
- [24] Z. M. Mao, C. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A precise and efficient evaluation of the proximity between web clients and their local DNS servers. In *USENIX Conference*, June 2002.
- [25] D. Mazières. A toolkit for user-level file systems. In USENIX Conference, June 2001.
- [26] A. Mislove, A. Post, A. Haeberlen, and P. Druschel. Experiences in building and operating a reliable peer-to-peer application. In *EuroSys*, Apr. 2006.
- [27] D. Morrison. Practical algorithm to retrieve information coded in alphanumeric. J. ACM, 15(4), Oct. 1968.
- [28] E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In INFOCOM, June 2002.
- [29] OASIS. http://www.coralcdn.org/oasis/, 2006.
- [30] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In SIGCOMM, Aug. 2001.
- [31] C. Patridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Network Working Group, Nov. 1993.
- [32] D. M. R. Periakaruppan and J. Donohoe. Where in the world is netgeo.caida.org? In INET, June 2000.
- [33] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on planetlab. In WORLDS, Dec. 2005.
- [34] PlanetLab. http://www.planet-lab.org/, 2005.
- [35] Quova. http://www.quova.com/, 2006.
- [36] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *IMW*, Nov. 2002.
- [37] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In SIGCOMM, Aug. 2005.
- [38] RouteViews. http://www.routeviews.org/, 2006.
- [39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In IFIP/ACM Middleware, Nov 2001.
- [40] K. Shanahan and M. J. Freedman. Locality prediction for oblivious clients. In *IPTPS*, Feb. 2005.
- [41] Sleepycat. BerkeleyDB v4.2, 2005.
- [42] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. In *IEEE/ACM Trans.* on Networking, 2002.
- [43] J. Stribling. PlanetLab AllPairsPing data, 08-03-2005:11:14:19. http://www.pdos.lcs.mit.edu/ strib/pl_app/, 2005.
- [44] W. Theilmann and K. Rothermel. Dynamic distance maps of the Internet. In *IEEE INFOCOM*, Mar 2001.
- [45] B. Wong and E. G. Sirer. ClosestNode.com: an open access, scalable, shared geocast service for distributed systems. SIGOPS OSR, 40(1), 2006.
- [46] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In SIG-COMM, Aug. 2005.

OverCite: A Distributed, Cooperative CiteSeer

Jeremy Stribling, Jinyang Li, Isaac G. Councill, M. Frans Kaashoek, Robert Morris

MIT Computer Science and Artificial Intelligence Laboratory

†New York University and MIT CSAIL, via UC Berkeley ††Pennsylvania State University

Abstract

CiteSeer is a popular online resource for the computer science research community, allowing users to search and browse a large archive of research papers. CiteSeer is expensive: it generates 35 GB of network traffic per day, requires nearly one terabyte of disk storage, and needs significant human maintenance.

OverCite is a new digital research library system that aggregates donated resources at multiple sites to provide CiteSeer-like document search and retrieval. OverCite enables members of the community to share the costs of running CiteSeer. The challenge facing OverCite is how to provide scalable and load-balanced storage and query processing with automatic data management. OverCite uses a three-tier design: presentation servers provide an identical user interface to CiteSeer's; application servers partition and replicate a search index to spread the work of answering each query among several nodes; and a distributed hash table stores documents and meta-data, and coordinates the activities of the servers.

Evaluation of a prototype shows that OverCite increases its query throughput by a factor of seven with a nine-fold increase in the number of servers. OverCite requires more total storage and network bandwidth than centralized CiteSeer, but spreads these costs over all the sites. OverCite can exploit the resources of these sites to support new features such as document alerts and to scale to larger data sets.

1 Introduction

Running a popular Web site is a costly endeavor, typically requiring many physical machines to store and process data and a fast Internet pipe to push data out quickly. It's

This research was conducted as part of the IRIS project (http://project-iris.net/), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660. Isaac G. Councill receives support from NSF SGER Grant IIS-0330783 and Microsoft Research.

common for non-commercial Web sites to be popular yet to lack the resources to support their popularity. Users of such sites are often willing to help out, particularly in the form of modest amounts of compute power and network traffic. Examples of applications that thrive on volunteer resources include SETI@home, BitTorrent, and volunteer software distribution mirror sites. Another prominent example is the PlanetLab wide-area testbed [36], which is made up of hundreds of donated machines over many different institutions. Since donated resources are distributed over the wide area and are usually abundant, they also allow the construction of a more fault tolerant system using a geographically diverse set of replicas.

In order to harness volunteer resources at many sites, a Web service needs a design that will increase its capacity as servers are added. This paper explores such a design in OverCite, a multi-site version of the CiteSeer repository of computer science research papers [30]. We choose CiteSeer for our study because of its value to the research community. However, despite its popularity, CiteSeer went mostly unmaintained after its initial development at NEC until a volunteer research group at Pennsylvania State University (PSU) took over the considerable task of running and maintaining the system.

1.1 Multi-site Web Design

Many designs and tools exist for distributing a Web service within a single-site cluster. The three-tier design is a common approach—a load-balancing front-end, application servers, and a shared back-end database—and services may also use techniques such as DDS [25], TACC [20], and MapReduce [17]. These solutions take advantage of reliable high-speed LANs to coordinate the servers in the cluster. Such solutions are less well suited to servers spread over the Internet, with relatively low speed and less reliable connectivity [3].

Existing approaches to multi-site services include mirroring, strict partitioning, caching, and more recently distributed hash tables (DHTs). Mirroring and strict parti-

tioning eliminate inter-site communication during ordinary operations. Mirroring is not attractive for storageintensive services: CiteSeer, for example, would require each mirror site to store nearly a terabyte of data. Partitioning the service among sites works only if each client request clearly belongs to a particular partition, but this is not true for keyword queries in CiteSeer; in addition, CiteSeer must coordinate its crawling activities among all sites. While content distribution networks such as Akamai [2], Coral [21], or CoDeeN [48] would help distribute static content such as the documents stored by CiteSeer, they would not help with the dynamic portion of the Cite-Seer workload: keyword searches, navigation of the graph of citations between papers, ranking papers and authors in various ways, and identification of similarity among papers. Similarly, as discussed in the related work (see Section 6), no existing DHT has been used for an application of the complexity of CiteSeer.

1.2 OverCite

What is needed is a design that parallelizes CiteSeer's operations over many sites to increase performance, partitions the storage to minimize the per-site burden, allows the coordination required to perform keyword searches and crawling, and can tolerate network and site failures. OverCite's design satisfies these requirements. Like many cluster applications, it has a three-tier design: multiple Web front-ends that accept queries and display results, application servers that crawl, generate indices, perform keyword searches on the indices, and a DHT back-end that aggregates the disks of the donated machines to store documents, meta-data, and coordination state.

The DHT back-end provides several distinct benefits. First, it is self-managing, balancing storage load automatically as volunteer servers come and go. Second, it handles replication and location of data to provide high availability. Finally, the DHT provides a convenient rendezvous service for producers and consumers of meta-data that must coordinate their activities. For example, once a node at one site has crawled a new document and stored it in the DHT, the document will be available to DHT nodes at other sites.

OverCite's primary non-storage activity is indexed keyword search, which it parallelizes with a scheme used in cluster-based search engines [7, 20]. OverCite divides the inverted index in k partitions. Each node stores a copy of one partition on its local disk; with n nodes, n/k nodes store a particular partition. OverCite sends a user query to k index servers, one for each partition, and aggregates the results from these index servers.

1.3 Contributions

The contributions of this paper are as follows¹:

- A three-tier DHT-backed design that may be of general use to multi-site Web services.
- OverCite, a multi-site deployment of CiteSeer.
- An experimental evaluation with 27 nodes, the full CiteSeer document set, and a trace of user queries issued to CiteSeer. A single-node OverCite can serve 2.8 queries/s (CiteSeer can serve 4.8 queries/s), and OverCite scales well: with 9 nodes serving as frontend query servers, OverCite can serve 21 queries/s.
- A case study of a challenging use of a DHT. OverCite currently stores 850 GB of data (amounting to tens of millions of blocks), consuming about a factor of 4 more total storage than CiteSeer itself. Each keyword query involves tens of DHT operations and is completed with reasonable latency.

We conjecture that OverCite's three-tier design may be useful for many Web services that wish to adopt a multisite arrangement. Services tend to consist of a mix of static content and dynamic operations. As long as the consistency requirements are not strict, a DHT can be used as a general-purpose back-end, both to hold inter-site coordination state and to spread the storage and serving load of large static documents.

1.4 Roadmap

Section 2 describes the design and operation of the current CiteSeer implementation. Section 3 gives the design of OverCite. Section 4 details the OverCite prototype implementation, and Section 5 evaluates the implementation. Section 6 discusses related work, and finally Section 7 concludes.

2 CiteSeer Background

CiteSeer [30] crawls, stores and indexes more than half a million research papers. CiteSeer's hardware consists of a pair of identical servers running the following components. A Web crawler visits a set of Web pages that are likely to contain links to PDF and PostScript files of research papers. If it sees a paper link it has not already fetched, CiteSeer fetches the file and parses it to extract text and citations. Then it applies heuristics to check if the

¹We presented a preliminary design, without an implementation, in an earlier workshop paper [45].

Number of papers	674,720
New documents per week	1000
HTML pages visited	113,000
Total document storage	803 GB
Avg. document size (all formats)	735 KB
Total meta-data storage	45 GB
Total inverted index size	22 GB
Hits per day	800,000
Searches per day	250,000
Total traffic per day	34 GB
Document traffic per day	21 GB
Avg. number of active conns	68
Avg. load per CPU	66%

Table 1: Statistics of the PSU CiteSeer deployment.

document duplicates an existing document; if not, it adds meta-data about the document to its tables. CiteSeer periodically updates its inverted index with newly crawled documents, indexing only the first 500 words of each document. The Web user interface accepts search terms, looks them up in the inverted index, and presents data about the resulting documents.

CiteSeer assigns a document ID (DID) to each document for which it has found a PDF or Postscript file, and a citation ID (CID) to every bibliography entry within a document. CiteSeer knows about many papers for which it has seen citations but not found a document file. For this reason CiteSeer assigns a "group ID" (GID) to each known paper for use in contexts where a file is not required. The GID also serves to connect newly inserted documents to previously discovered citations. All the IDs are numerically increasing 14-byte numbers.

CiteSeer stores the PDF/PostScript of each research paper (as well as the ASCII text extracted from it) in a local file system. In addition, CiteSeer stores the following meta-data tables to help identify papers and filter out possible duplicates:

- The document meta-data table (Docs), indexed by DID, which records each document's authors, title, year, abstract, GID, CIDs of the document's citations, number of citations to the document, etc.
- The citation meta-data table (Cites), indexed by CID, which records each citation's GID and citing document DID.
- A table (Groups) mapping each GID to the corresponding DID (if a DID exists) and the list of CIDs that cite it.
- A table indexed by the checksum of each fetched document file, used to decide if a file has already been processed.

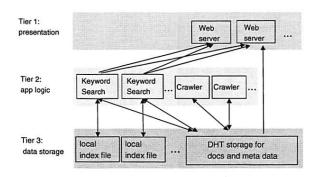


Figure 1: Overview of OverCite's three-tier architecture. The modules at all tiers are run at many (if not all) nodes. The Tier 1 Web server interacts with multiple search engines on different nodes to perform a single user keyword search. The Tier 2 modules use the underlying DHT to store documents and corresponding meta-data. The keyword search engine also stores data locally outside the DHT.

- A table indexed by the hash of every sentence Cite-Seer has seen in a document, used to gauge document similarity.
- A table (URLs) to keep track of which Web pages need to be crawled, indexed by URL.
- A table (Titles) mapping paper titles and authors to the corresponding GID, used to find the target of citations observed in paper bibliographies.

Table 1 lists statistics for the deployment of CiteSeer at PSU as of September 2005. The CiteSeer Web site uses two servers each with two 2.8 GHz processors. The two servers run independently of each other and each has a complete collection of PDF/Postscript documents, inverted indices, and meta-data. Most of the CPU time is used to satisfy keyword searches and to convert document files to user-requested formats.. The main costs of searching are lookups in the inverted index, and collecting and displaying meta-data about search results.

3 Design

A primary goal of OverCite's design is to ensure that its performance increases as volunteers contribute nodes. OverCite addresses this challenge with a three-tier design (see Figure 1), similar to cluster-based Web sites, but with the database replaced with a DHT. The modules in Tier 1 accept keyword queries and aggregate results from Tier 2 modules on other nodes to present the traditional CiteSeer interface to users. The Tier 2 modules perform keyword searches and crawling for new documents. The Tier 1 and 2 modules use the DHT servers in Tier 3 to store and fetch

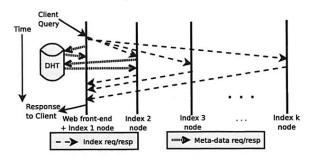


Figure 2: The timeline of a query in OverCite, and the steps involved. Each vertical bar represents a node with a different index partition. DHT meta-data lookups are only required at index servers without cached copies of result meta-data.

the documents, meta-data, and shared state for coordination. When a site contributes one or more nodes to the system, each node starts a DHT server to put the donated disk space and network bandwidth to use.

3.1 Overview: The Life of a Query

Figure 2 depicts the timeline of a query. A subset of OverCite nodes run a Web user interface, using round-robin DNS or OASIS [22] to spread the client load. The front-end accepts a query with search terms from the user and uses a scheme similar to the ones used by cluster-based search engines to parallelize the search: the front-end sends the query to k index servers, each responsible for 1/kth of the index.

Given a keyword query, each index server searches for the m most highly-ranked documents in its local index; the search engine ranks documents by the relative frequency and position of the query keywords within the text as well as by citation count. The index server looks up the meta-data for these m documents in the DHT to supply additional information like citation counts back to the front-end. Servers cache this meta-data locally to speed up future searches. The front-end is responsible for merging the results from the k index servers and displaying the top m to the user.

3.2 Global Data Structures

OverCite stores document files and meta-data in a DHT shared among all the nodes. Table 2 lists the data tables that OverCite stores in the DHT. In addition to existing CiteSeer data structures, Crawl and URLs tables are added to coordinate distributed crawling activity as explained in Section 3.4.

OverCite needs the following four properties from the

Name	Key	Value
Docs	DID	FID, GID, CIDs, etc.
Cites	CID	DID, GID
Groups	GID	DID + CID list
Shins	hash(shingle)	list of DIDs
Crawl		list of page URLs
URLs	hash(doc URL)	date file last fetched
Titles	hash(Ti+Au)	GID
Files	FID	Document file

Table 2: The data structures OverCite stores in the DHT. The Files table stores immutable content hash blocks for PDF/Postscript documents, indexed by a root content hash key FID. All the rest of the tables are stored as append-only blocks.

underlying DHT. First, the DHT should be able to store a large amount of data with a put/get interface, where a block is named by the hash of the block's content. To balance the storage of data well across the nodes, OverCite's blocks are at most 16 KB in size. OverCite stores large files, such as PDF and PostScript files, as a Merkle tree of content-hash blocks [35]; the file's identifier, or FID, is the DHT key of the root block of the Merkle tree.

Second, the DHT must support append-only blocks [38]. OverCite stores each entry in the metadata tables (Docs, Cites, and Groups) listed in Table 2 as an append-only block, indexed using a randomly-generated 20-byte DID, CID or GID. OverCite treats the append-only blocks as an update log, and reconstructs the current version of the data by applying each block of appended data as a separate update. Append-only logs can grow arbitrarily large as a document's meta-data is updated, but in practice is usually small.

OverCite uses append-only blocks, rather than using fully-mutable blocks, because append-only blocks simplify keeping data consistent. Any OverCite node can append to an append-only block at any time, and the DHT ensures that eventually all replicas of the block will see all appended data (though strict order is not necessarily enforced by the DHT).

Third, the DHT should try to keep the data available (perhaps by replicating it), although ultimately OverCite can regenerate it by re-crawling. Furthermore, the DHT should be resilient in the face of dynamic membership changes (churn), though we do no expect this to be a major issue in a managed, cooperative system like OverCite. Finally, the DHT should support quick (preferably one-hop) lookups in systems with a few hundred nodes.

3.3 Local Data Structures

Each OverCite node stores data required for it to participate in keyword searches on its local disk. This local data includes an inverted index yielding the list of documents containing each word and extracted ASCII text for each document in the inverted index that is used to present the context around each search result.

OverCite partitions the keyword index by document, so that each node's inverted index includes documents from only one partition. Partitioning reduces the index storage requirement at each node, and reduces latency when load is low by performing each query in parallel on multiple nodes. OverCite typically has fewer index partitions than nodes, and replicates each partition on multiple nodes. Replication increases fault-tolerance and increases throughput when the system is busy since it allows different queries to be processed in parallel. This index partitioning and replication strategy is used by clusterbased search engines such as Google [7] and HotBot [20]. Compared with several other peer-to-peer search proposals [31, 37, 46, 47], partitioning by document is bandwidth efficient, load-balanced, and provides the same quality of results as a centralized search engine.

OverCite uses k index partitions, where k is less than the number of nodes (n). Each node stores and searches one copy of one index partition; if there are n nodes, there are n/k copies of each index partition. The front-end sends a copy of each query to one server in each partition. Each of the k servers processes the query using 1/k'th of the full index, which requires about 1/k'th the time needed to search a complete index.

A large k decreases query latency at low load due to increased parallelism, and may also increase throughput since a smaller inverted index is more likely to fit in each node's disk cache. However, a large k also increases network traffic, since each node involved in a query returns about 170 bytes of information about up to m of its best matches. Another reason to restrict k is that the overall keyword search latency may be largely determined by the response time of the slowest among the k-1 remote index servers. This effect is somewhat mitigated because the front-end sends each query to the lowest-delay replica of each index partition. We also plan to explore forwarding queries to the least-loaded index partition replicas among nearby servers.

A node's index partition number is its DHT identifier mod k. A document's index partition number is its DID mod k. When the crawler inserts a new document into the system, it notifies at least one node in the document's partition; the nodes in a partition periodically exchange notes about new documents.

3.4 Web Crawler

The OverCite crawler design builds on several existing proposals for distributed crawling (e.g., [9, 12, 32, 41]). Nodes coordinate the crawling effort via a list of to-becrawled Web page URLs stored in the DHT. Each crawler process periodically chooses a random entry from the list and fetches the corresponding page.

For each link to a Postscript or PDF file a node finds on a Web page, the crawler performs a lookup in the URLs table to see whether the document has already been downloaded. After the download, the crawler parses the file - extracting meta-data (e.g., title, authors, citations, etc.) as well as the bare ASCII text of the document and checks whether this is a duplicate document. This requires (1) looking up the FID of the file in Files; (2) searching for an existing document with the same title and authors using Titles; and (3) verifying that, at a shingle level, the document sufficiently differs from others. OverCite uses shingles [8] instead of individual sentences as in CiteSeer for duplicate detection. Checking for duplicates using shingles is effective and efficient, resulting in a small Shins table. If the document is not a duplicate, the crawler inserts the document into Files as Postscript or PDF. The node also updates Docs, Cites, Groups, and Titles to reflect this document and its meta-data. The crawler puts the extracted ASCII text in the DHT and informs one index server in the document's partition of newly inserted document's DID.

While many enhancements to this basic design (such as locality-based crawling and more intelligent URL partitioning) are both possible and desirable, we defer optimizations of the basic crawler design to future work. Crawling and fetching new documents will take approximately three times more bandwidth than CiteSeer uses in total, spread out among all the servers. We have shown these calculations in previous work [45].

4 Implementation

The OverCite implementation consists of several software modules, corresponding to the components described in Section 3. The current implementation does not yet include a *Crawler* module; we have populated OverCite with existing CiteSeer documents. The OverCite implementation consists of over 11,000 lines of C++ code, and uses the SFS libasync library [34] to provide an event-driven, single-threaded execution environment for each module. Figure 3 shows the overall interactions of different OverCite modules with each other and the DHT. Modules on the same machine communicate locally through

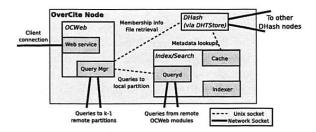


Figure 3: Implementation overview. This diagram shows the communication paths between OverCite components on a single node, and network connections between nodes.

Unix domain sockets; modules on different machines communicate via TCP sockets. All inter-module communication occurs over the Sun RPC protocol.

The *OCWeb* Module. The *OCWeb* module provides a Web interface to accept keyword queries and display lists of matching documents. *OCWeb* is implemented as a module for the OK Web Server (OKWS) [29], a secure, high-performance Web server. Because OKWS also uses libasync, this choice of Web server allows *OCWeb* to access the *DHTStore* library and interact with DHash directly.

The *DHTStore* Module. The *DHTStore* module acts as an interface between the rest of the OverCite system and the DHT. The module takes the form of a library that can be used by other system components to retrieve meta-data and documents. The implementation uses the DHash [16] DHT. Each OverCite node runs three DHash virtual nodes per physical disk to balance the storage load evenly.

Meta-data Storage. OverCite stores its tables (see Table 2) in the the DHT's single 20-byte key space, with each row of each table represented as a DHT data block. A table row's DHT key is the hash of the corresponding OverCite identifier (e.g. DID, CID, etc.) concatenated with the table's name; for example, the keys used to index the Docs table are the hash of the string "Docs" concatenated with the DID of the document. The hashing spreads the meta-data evenly over the DHT nodes. OverCite appends data to some kinds of table rows; for example, OverCite appends to a Groups block when it finds a new citation to a document. The current implementation supports the tables listed in Table 2 except Shins, Crawl, and URLs.

The Index/Search Module. The Index/Search module consists of a query server daemon (Queryd), a meta-

data and text-file cache, and an index generator (*Indexer*). We chose to implement our own search engine instead of reusing the current built-in CiteSeer search engine, because we have found it difficult to extend CiteSeer's search engine to include new functions (e.g., to experiment with different ranking functions).

The *Indexer* periodically retrieves the ASCII text and meta-data for new documents to be included in the node's index partition from the DHT and caches them on the local disk. It updates the local inverted index file based on the local disk cache. The inverted index consists of posting lists of document numbers and the ASCII file offset pairs for each word. Compared to CiteSeer's inverted index structure, *Indexer*'s inverted index optimizes query speed at the cost of slower incremental index updates. The *Indexer* indexes the first 5000 words of each document (CiteSeer indexes the first 500). The document and offset pairs in a posting list are ranked based on the corresponding citation counts.

Upon receiving a query, Queryd obtains the list of matching documents by intersecting the posting lists for different keywords. On-disk posting lists are mmap-ed into the memory, causing them to be paged in the buffer cache. Queryd scores each result based on the document ranks, the file offsets where a keyword occurs, and the proximity of keywords in the matching document. Queryd returns the top m scored documents as soon as it judges that no further lower ranked documents can generate higher scores than the existing top m matches.

For each of the top m matches, Queryd obtains the context (the words surrounding the matched keywords) from the on-disk cache of ASCII files. In parallel, it also retrieves the corresponding meta-data from the DHT. Upon completion of both context and meta-data fetches, Queryd returns the results to either the local Query manager or the remote Queryd.

5 Evaluation

This section explores how OverCite scales with the total number of nodes. Although the scale of the following preliminary experiments is smaller than the expected size of an OverCite system, and the code is currently an early prototype, this evaluation demonstrates the basic scaling properties of the OverCite design. We plan to perform a more in-depth analysis of the system once the code base matures and more nodes become available for testing.

5.1 Evaluation Methods

We deployed OverCite on 27 nodes: sixteen at MIT and eleven spread over North America, most of which are part of the RON test-bed [3]. The DHash instance on each node spawned three virtual nodes for each physical disk to balance load; in total, the system uses 47 physical disks. These disks range in capacity from 35 GB to 400 GB.

We inserted the 674,720 documents from the Cite-Seer repository into our OverCite deployment, including the meta-data for the documents, their text and Postscript/PDF files, and the full citation graph between all documents. CiteSeer uses several heuristics to determine whether a document in its repository is a duplicate of another, and indexes only non-duplicates; OverCite indexes the same set of non-duplicate documents (522,726 documents in total). OverCite currently stores only the *original* copy of each document (*i.e.*, the document originally discovered and downloaded by the crawler), while CiteSeer stores Postscript, compressed Postscript, and PDF versions for every document. We plan to store these versions as well in the near future.

Unless otherwise stated, each document is indexed by its first 5000 words, and each experiment involves two index partitions (k=2). All Queryd modules return up to 20 results per query (m=20), and the context for each query contains one highlighted search term. Furthermore, each node has a complete on-disk cache of the text files for all documents in its index partition (but not the document meta-data or Postscript/PDF files). The results represent the average of five trials for each experiment.

To evaluate the query performance of OverCite, we used a trace of actual CiteSeer queries, collected in October 2004. The client machine issuing queries to OverCite nodes is a local MIT node that is not participating in OverCite, and that can generate requests concurrently to emulate many simultaneous clients.

5.2 Query Throughput

One of the chief advantages of a distributed system such as OverCite over its centralized counterparts is the degree to which OverCite uses resources in parallel. In the case of OverCite, clients can choose from many different Web servers (either manually or through DNS redirection), all of which have the ability to answer any query using different sets of nodes. Because each index partition is replicated on multiple nodes, OverCite nodes have many forwarding choices for each query. We expect that if clients issue queries concurrently to multiple servers, each of which is using different nodes as index neighbors, we will achieve a corresponding increase in system-wide

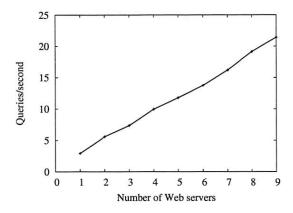


Figure 4: Average query throughput on a distributed OverCite system, as a function of the number of Web (Tier 1) servers. The client issues 128 concurrent queries at a time.

throughput. However, because the nodes are sharing (and participating in) the same DHT, their resources are not entirely independent, and so the effect on throughput of adding Tier 1 and 2 servers is non-obvious.

To evaluate scalability, we measured throughput with varying numbers of front-end nodes. The value of k was 2, and the total number of nodes in each configuration is twice the number of front ends. Each front-end is paired with a non-front-end holding the other partition. A client at MIT keeps 128 queries active, spread among all of the available Web servers (randomly choosing which Web server gets which query). The test uses servers chosen at random for each experiment trial.

Figure 4 shows the number of queries per second processed by OverCite, as a function of the number of frontend servers. Adding additional front-end servers linearly increases the query throughput. With a single front-end server OverCite serves about 3 queries per second, while 9 front-end servers satisfy 21 queries per second. Despite the fact that the servers share a common DHT (used when looking up document meta-data), the resources of the different machines can be used by OverCite to satisfy more queries in parallel.

For comparison, a single-server CiteSeer can process 4.8 queries per second with the same workload but slightly different hardware. CiteSeer indexes only the first 500 words per document. The corresponding single-node throughput for OverCite is 2.8 queries per second.

5.3 Performance Breakdown

This subsection evaluates the latency of individual queries. The *Queryd* daemon on a node performs three

Context				
k	k Search Context		DHT wait	Total
2	118.14	425.11 (37.29 each)	21.27	564.52
4	78.68	329.04 (32.97 each)	23.97	431.69

	No context			
k	Search	DHT wait	Total	
2	53.31	170.03	223.34	
4	28.07	208.30	236.37	

Table 3: Average latencies (in ms) of OverCite operations for an experiment that generates context results, and one that does not give context results. **Search** shows the latency of finding results in the inverted index; **Context** shows the latency of generating the context for each result (the per-result latency is shown in parentheses); **DHT wait** shows how long OverCite waits for the last DHT lookup to complete, *after* the search and context operations are completed. Only one query is outstanding in the system.

main operations for each query it receives: a search over its inverted index, the generation of context information for each result, and a DHT meta-data lookup for each result. All DHT lookups happen in parallel, and context generation happens in parallel with the DHT lookups.

Table 3 summarizes the latencies of each individual operation, for experiments with and without context generation. **Search** shows the latency of finding results in the inverted index; **Context** shows the latency of generating the context for each result; **DHT wait** shows how long OverCite waits for the last DHT lookup to complete, *after* the search and context operations are completed. The table gives results for two different values of k, 2 and 4.

The majority of the latency for a single query comes from context generation. For each result the server must read the ASCII text file from disk (if it is not cached in memory), which potentially involves several disk seeks: OverCite currently organizes the text files in a two-level directory structure with 256 directories per level. In the future, we plan to explore solutions that store all the ASCII text for all documents in a single file, to avoid the overhead of reading the directory structures. Because context generation dominates the search time, and DHT lookups occur in parallel with the context generation, OverCite spends little time waiting for the DHT lookups to return; less than 4% of the total latency is spent waiting for and processing the meta-data lookups.

Increasing k decreases the latency of the search over the inverted index, because each partition indexes fewer documents and the size of the inverted index largely determines search speed. Interestingly, search latency is significantly lower when OverCite does not generate context,

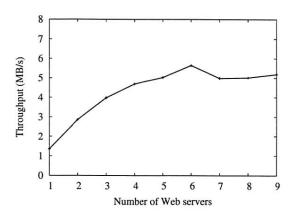


Figure 5: Average throughput of OverCite serving Postscript/PDF files from the DHT, as a function of the number of front-end Web servers.

presumably due to more efficient use of the buffer cache for the inverted index when the ASCII text files are not read.

Without context generation, the meta-data lookups become a greater bottleneck to the latency, and in fact increasing k causes DHT lookups to slow down. With a larger k value, there are more total lookups happening in the system, since each node retrieves its top m results.

5.4 File Downloads

This section evaluates how well OverCite serves PDF and Postscript documents. We measured the rate at which a single front-end at CMU could serve documents to the client at MIT. The client kept 128 concurrent requests active. The network path from CMU to MIT is capable of carrying 11.4 megabytes/second, as measured with ttcp using UDP.

OverCite's document throughput averaged 1.3 megabytes/second. The download rate from the single front-end is limited by the rate that the server can download blocks from the DHT, which is determined by the bottleneck bandwidth between the front-end and the slowest node in the system [16]. The measured UDP network capacity over the slowest access link was 1.14 MB/s.

If the client uses multiple front-ends to download files, it can achieve a higher throughput. Figure 5 shows the file-serving throughput of OverCite as a function of the number of front-end Web servers used by the client, when the client requests a total of 128 files concurrently from the system. The throughput plateaus at nearly five times the throughput from a single server, and is similar to the throughput for the same number of servers measured in

Property	Cost
Document/meta-data storage	270 GB
Index size	14 GB
Total storage	284 GB

Table 4: Storage statistics for a centralized server.

a previous evaluation of DHash [16]. Table 1 shows that currently CiteSeer serves only 35 GB/day (or 425 KB/s), a load that our OverCite implementation can easily handle.

5.5 Storage

Finally, we compare the storage costs of OverCite to those of a centralized solution. The centralized solution keeps one copy of the original crawled file (as noted in Section 5.1), the extracted ASCII text, and the full set of CiteSeer meta-data for each document, in addition to a full inverted index built using OverCite's *Indexer* module. We compare this to the storage costs measured on our 27-node, 47-disk, 2-partition OverCite deployment. OverCite storage costs do not include the on-disk cache of ASCII text files used to generate context information; these files are included in the DHT storage costs, and the cache can always be created by downloading the files from the DHT.

Table 4 shows the storage costs of the centralized solution. The total space used is 284 GB, the majority of which is documents and meta-data. Table 5 shows the average per-node storage costs measured on our OverCite deployment. The system-wide storage cost is 1034.3 GB. An individual node in the system with one physical disk and one *Index/Search* module has a cost of 24.9 GB.

If we assume that each individual disk is its own node and has its own copy of the index, the full OverCite system would use 4.1 times as much space as the centralized solution. Given that OverCite's DHash configuration uses a replication factor of 2, this overhead is higher than expected. Some blocks are actually replicated more than twice, because DHash does not delete old copies of blocks when copying data to newly-joined nodes. Storing many small blocks in the database used by DHash also incurs overhead, as does OverCite's Merkle tree format for storing files.

For our current implementation, adding 47 nodes to the system decreased the *per-node* storage costs by about a factor of 11.4; assuming this scaling factor holds indefinitely, adding n nodes to the system would decrease pernode storage costs by a factor of roughly n/4. Therefore, we expect that an OverCite network of n nodes can handle n/4 times as many documents as a single CiteSeer node.

Property	Individual cost	System cost
Document/meta-data	18.1 GB	18.1 GB × 47
storage		= 850.7 GB
Index size	6.8 GB	$6.8 \mathrm{GB} \times 27$
		= 183.6 GB
Total storage	24.9 GB	1034.3 GB

Table 5: Average per-node storage statistics for the OverCite deployment. There are 27 nodes (and 47 disks) in the system.

6 Related Work

Many digital libraries exist. Professional societies such as ACM [1] and IEE [27] maintain online repositories of papers published at their conferences. Specific academic fields often have their own research archives, such as arXiv.org [5], Google Scholar [24], and CiteSeer [30], which allow researchers to search and browse relevant work, both new and old. More recently, initiatives like DSpace [43] and the Digital Object Identifier system [18] seek to provide long-term archival of publications. The main difference between these systems and OverCite is that OverCite is a community-based initiative that can incorporate donated resources at multiple sites across the Internet.

Previous work on distributed library systems includes LOCKSS [39], which consists of many persistent web caches that can work together to preserve data for decades against both malicious attacks and bit rot. Furthermore, the Eternity Service [4] uses peer-to-peer technology to resist censorship of electronic documents. There have also been a number of systems for searching large data sets [6, 11,23,26,33,40,47,49] and crawling the Web [9,12,32,41] using peer-to-peer systems. We share with these systems a desire to distribute work across many nodes to avoid centralized points of failure and performance bottlenecks.

Services like BitTorrent [14] and Coral [21] provide an alternative style of content distribution to a DHT. Like DHTs such as DHash [16], these systems can find the closest copy of a particular data item for a user, and can fetch many data items in parallel. However, the DHT-based three tier design is more closely aligned with Web services that have both dynamic and static content.

DHTs have been used in many applications (e.g., [15, 19, 44]), but few have required substantial storage, or perform intensive calculations with the data stored. PIER [26] is a distributed query engine for wide-area distributed systems, and extends the DHT with new operators. The Place Lab Web service [10] is an investigation into how an unmodified DHT can be used to implement complex functions such as range-queries, but computes with little data (1.4 million small records). Usenet-

DHT [42] stores a substantial amount of data but doesn't require computation on the data. Because these applications are simpler than CiteSeer, they do not require the three-tier design used in this paper.

7 Conclusion and Future Work

Using a three-tier design, OverCite serves more queries per second than a centralized server, despite the addition of DHT operations and remote index communication. Given the additional resources available with OverCite's design, a wider range of features will be possible; in the long run the impact of new capabilities on the way researchers communicate may be the main benefit of a more scalable CiteSeer.

For example, as the field of computer science grows, it is becoming harder for researchers to keep track of new work relevant to their interests. OverCite could help by providing an *alert* service to e-mail a researcher whenever a paper entered the database that might be of interest. Users could register queries that OverCite would run daily (e.g., alert me for new papers on "distributed hash table" authored by "Druschel"). This service clearly benefits from the OverCite DHT infrastructure as the additional query load due to alerts becomes distributed over many nodes. A recent proposal [28] describes a DHT-based alert system for CiteSeer. Other possible features include Amazon-like document recommendations, plagiarism detection, or including a more diverse range of documents, such as preprints or research from other fields.

Since OverCite's architecture allows it to include new resources as they become available, it can scale its capacity to meet the demands of imaginative programmers. We plan to create an open programming interface to the OverCite data (similar to CiteSeer's OAI interface [13]), allowing the community to implement new features and services on OverCite such as those listed above. We plan to launch OverCite as a service for the academic community in the near future to encourage these possibilities.

Acknowledgments

We thank Frank Dabek, Max Krohn, and Emil Sit for their tireless help debugging and discussing; David Karger and Scott Shenker for formative design conversations; Dave Andersen, Nick Feamster, Mike Howard, Eddie Kohler, Nikitas Liogkas, Ion Stoica, and all the RON host sites for volunteering machines, bandwidth, and time; the reviewers for their insightful comments; and C. Lee Giles for his continued support at PSU.

References

- [1] The ACM Digital Library. http://portal.acm.org/dl.cfm.
- [2] Akamai technologies, inc. http://www.akamai.com.
- [3] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Resilient overlay networks. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01) (2001).
- [4] ANDERSON, R. J. The Eternity Service. In Proceedings of the 1st International Conference on the Theory and Applications of Cryptology (1996).
- [5] arXiv.org e-Print archive. http://www.arxiv.org.
- [6] BAWA, M., MANKU, G. S., AND RAGHAVAN, P. SETS: Search enhanced by topic segmentation. In *Proceedings of the 2003 SIGIR* (July 2003).
- [7] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems 30 (1998).
- [8] BRODER, A. Z. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity of Sequences (June 1997).
- [9] BURKARD, T. Herodotus: A peer-to-peer web archival system. Master's thesis, Massachusetts Institute of Technology, May 2002.
- [10] CHAWATHE, Y., RAMABHADRAN, S., RATNASAMY, S., LAMARCA, A., SHENKER, S., AND HELLERSTEIN, J. A case study in building layered DHT applications. In *Proceedings of the* 2005 SIGCOMM (Aug. 2005).
- [11] CHAWATHE, Y., RATNASAMY, S., BRESLAU, L., LANHAM, N., AND SHENKER, S. Making Gnutella-like P2P systems scalable. In *Proceedings of the 2003 SIGCOMM* (Aug. 2003).
- [12] CHO, J., AND GARCIA-MOLINA, H. Parallel crawlers. In Proceedings of the 2002 WWW Conference (May 2002).
- [13] CiteSeer.PSU Open Archive Initiative Protocol. http://citeseer.ist.psu.edu/oai.html.
- [14] COHEN, B. Incentives build robustness in BitTorrent. In Proceedings of the Workshop on Economics of Peer-to-Peer Systems (June 2003).
- [15] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01) (Oct. 2001).
- [16] DABEK, F., KAASHOEK, M. F., LI, J., MORRIS, R., ROBERT-SON, J., AND SIT, E. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st NSDI* (Mar. 2004).
- [17] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec. 2004).
- [18] The Digital Object Identifier system. http://www.doi.org.
- [19] DRUSCHEL, P., AND ROWSTRON, A. PAST: Persistent and anonymous storage in a peer-to-peer networking environment. In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII) (May 2001), pp. 65–70.
- [20] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In Proceedings of the 16th ACM Symposium on Operating Systems Principles (1997).

- [21] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with Coral. In *Proceedings of the 1st NSDI* (Mar. 2004).
- [22] FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIÈRES, D. OASIS: Anycast for any service. In Proceedings of the 3rd NSDI (May 2006).
- [23] GNAWALI, O. D. A keyword set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, June 2002.
- [24] Google Scholar. http://scholar.google.com.
- [25] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for Internet service construction. In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000) (Oct. 2000).
- [26] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In *Proceedings of the 19th VLDB* (Sept. 2003).
- [27] Inspec. http://www.iee.org/Publish/INSPEC/.
- [28] KANNAN, J., YANG, B., SHENKER, S., SHARMA, P., BANER-JEE, S., BASU, S., AND LEE, S. J. SmartSeer: Using a DHT to process continuous queries over peer-to-peer networks. In *Pro*ceedings of the 2006 IEEE INFOCOM (Apr. 2006).
- [29] KROHN, M. Building secure high-performance web services with OKWS. In Proceedings of the 2004 Usenix Technical Conference (June 2004).
- [30] LAWRENCE, S., GILES, C. L., AND BOLLACKER, K. Digital libraries and autonomous citation indexing. *IEEE Computer 32*, 6 (1999), 67–71. http://www.citeseer.org.
- [31] LI, J., LOO, B. T., HELLERSTEIN, J. M., KAASHOEK, M. F., KARGER, D., AND MORRIS, R. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd IPTPS* (Feb. 2003).
- [32] LOO, B. T., COOPER, O., AND KRISHNAMURTHY, S. Distributed web crawling over DHTs. Tech. Rep. UCB//CSD-04-1332, UC Berkeley, Computer Science Division, Feb. 2004.
- [33] LOO, B. T., HUEBSCH, R., STOICA, I., AND HELLERSTEIN, J. M. The case for a hybrid P2P search infrastructure. In Proceedings of the 3rd IPTPS (Feb. 2004).
- [34] MAZIÈRES, D. A toolkit for user-level file systems. In Proceedings of the 2001 Usenix Technical Conference (June 2001).
- [35] MERKLE, R. C. A digital signature based on a conventional encryption function. In CRYPTO '87: Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (1988), pp. 369–378.
- [36] PlanetLab: An open platform for developing, deploying and accessing planetary-scale services. http://www.planet-lab.org
- [37] REYNOLDS, P., AND VAHDAT, A. Efficient peer-to-peer keyword searching. In Proceedings of the 4th International Middleware Conference (June 2003).
- [38] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RAT-NASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: A public DHT service and its uses. In *Proceedings of the 2005 SIGCOMM* (Aug. 2005).
- [39] ROSENTHAL, D. S. H., AND REICH, V. Permanent web publishing. In Proceedings of the 2000 USENIX Technical Conference, Freenix Track (June 2000).

- [40] SHI, S., YANG, G., WANG, D., YU, J., QU, S., AND CHEN, M. Making peer-to-peer keyword searching feasible using multi-level partitioning. In *Proceedings of the 3rd IPTPS* (Feb. 2004).
- [41] SINGH, A., SRIVATSA, M., LIU, L., AND MILLER, T. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *Proceedings of the SIGIR 2003 Workshop on Dis*tributed Information Retrieval (Aug. 2003).
- [42] SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead Usenet server. In Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (Feb. 2004).
- [43] SMITH, M. Dspace for e-print archives. High Energy Physics Libraries Webzine, 9 (Mar. 2004). http://dspace.org.
- [44] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In ACM SIG-COMM (Aug. 2002).
- [45] STRIBLING, J., COUNCILL, I. G., LI, J., KAASHOEK, M. F., KARGER, D. R., MORRIS, R., AND SHENKER, S. OverCite: A cooperative digital research library. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems* (Feb. 2005).
- [46] SUEL, T., MATHUR, C., WU, J.-W., ZHANG, J., DELIS, A., KHARRAZI, M., LONG, X., AND SHANMUGASUNDARAM, K. ODISSEA: A peer-to-peer architecture for scalable web search and information retrieval. In *Proceedings of the International Work-shop on the Web and Databases* (June 2003).
- [47] TANG, C., AND DWARKADAS, S. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of the 1st NSDI* (Mar. 2004).
- [48] WANG, L., PARK, K., PANG, R., PAI, V. S., AND PETERSON, L. Reliability and security in the codeen content distribution network. In Proceedings of the USENIX 2004 Annual Technical Conference (June 2004).
- [49] YANG, B., AND GARCIA-MOLINA, H. Improving search in peerto-peer networks. In *Proceedings of the 22nd ICDCS* (July 2002).

Colyseus: A Distributed Architecture for Online Multiplayer Games

Ashwin Bharambe

Carnegie Mellon University

ashu+@cs.cmu.edu

Jeffrey Pang
Carnegie Mellon University
jeffpang+@cs.cmu.edu

Srinivasan Seshan

Carnegie Mellon University

srini@cmu.edu

Abstract

This paper presents the design, implementation, and evaluation of Colvseus, a distributed architecture for interactive multiplayer games. Colyseus takes advantage of a game's tolerance for weakly consistent state and predictable workload to meet the tight latency constraints of game-play and maintain scalable communication costs. In addition, it provides a rich distributed query interface and effective pre-fetching subsystem to help locate and replicate objects before they are accessed at a node. We have implemented Colyseus and modified Quake II, a popular first person shooter game, to use it. Our measurements of Quake II and our own Colyseus-based game with hundreds of players shows that Colyseus effectively distributes game traffic across the participating nodes, allowing Colyseus to support low-latency game-play for an order of magnitude more players than existing single server designs, with similar per-node bandwidth costs.

1 Introduction

USENIX Association

Networked games are rapidly evolving from small 4-8 person, one-time play games to large-scale games involving thousands of participants and persistent game worlds. Almost all networked games, however, are centralized - players send control messages to a central server and the server sends relevant state updates to all active players. This approach suffers from the well known robustness and scalability problems of single server designs. For example, high update rates prevent even well provisioned servers from supporting more than several tens of players in first person shooter (FPS) games. Further, client-server game designs often force players to rely on infrastructure provided by the game manufacturers. These infrastructures are sometimes not well provisioned nor long-lived; thus, they either provide poor performance or prevent users from playing their game long after their purchase.

A distributed design can potentially address the above shortcomings. However, architecting a distributed application is difficult due to the challenges of partitioning the application's state (e.g., game objects) and execution (e.g., the logic to simulate player and game AI actions) among the participating nodes. Distributing a

networked game is even more difficult due to the performance demands of real-time game-play. In addition, since the game-play of an individual player translates to updates to the shared state of the game application, there is much more write traffic and write-sharing than most distributed applications.

Fortunately, we can take advantage of two fundamental properties of games to address these challenges. First, games tolerate weak consistency in the application state. For example, current client-server implementations minimize interactive response time by presenting a weakly consistent view of the game world to players. Second, game-play is usually governed by a strict set of rules that make the reads and writes to the shared state highly predictable. For example, most reads and writes by a player occur upon objects that are physically close to that player in the game world. The challenge, then, is to arrive at a scalable and efficient state and logic partitioning that enables reasonably consistent, low-latency game-play. This paper presents the design, implementation, and evaluation of Colyseus, a novel distributed architecture for interactive multiplayer games designed to achieve the above goals.

In Colyseus, any node may create read-only replicas of any game object. However, objects in Colyseus follow a single-copy consistency model — i.e., all updates to an object are serialized through exactly one primary copy in the system. This approach mirrors the consistency model of existing client-server architectures on a per object basis. Although replicas are only kept weakly consistent with the primary copy, they enable the low-latency read access needed to keep game execution timely. The challenge is for each node to determine the set of replicas it needs in advance of executing any game logic. Colyseus provides a rich query interface over the system-wide collection of objects to identify and fetch required objects. We have implemented this query interface on both a randomized distributed hash table (DHT) [28] and a dynamically load balanced, rangebased DHT [3]. However, lookups in DHTs can be too slow for finding required replicas in games. To hide this lookup latency, Colyseus uses locality and predictability in data access patterns to speculatively pre-fetch objects. This mechanism is only used to discover relevant objects; updates are propagated from primary copies to replicas directly. We show that the combination of all these techniques is critical to enabling interactive gameplay.

Colyseus enables games to efficiently use widely distributed servers to support a large community of users. We have modified Quake II [22], a popular serverbased First Person Shooter (FPS) game, to run on our implementation of Colyseus, and have also used measurements of Quake III [23] game-play to develop our own Colyseus-based game with players that mimic the Quake III workload. These concrete case studies illustrate the practicality of using our architecture to distribute existing game implementations. Our measurements on an Emulab testbed with hundreds of players show that Colyseus is effective at distributing game traffic and workload across the participating nodes, while providing servers and players with low-latency and consistent views of the game world. In the following sections, we provide background about general game design as well as the design and evaluation of Colyseus.

2 Background

In this section, we survey the requirements of online multiplayer games and demonstrate the fundamental limitations of existing client-server implementations. In addition, we provide evidence that resources exist for distributed deployments of multiplayer games.

2.1 Contemporary Game Design

To determine the requirements of multiplayer games, we studied the source code of several popular and publicly released engines for *virtual reality* games, including Quake II [22], Quake III [23], and the Torque Networking Library [30]. In these games, each *player* (game client) controls an *avatar* (player's representative in the game) in a large *game world*, though a player only interacts with a small portion of the world at any given time. This description applies to many popular genres, including FPSs (such as *Quake* and *Counter Strike*), role playing games (RPGs) (such as *Everquest* and *World of Warcraft*), and others. There are certainly some game genres that do not fit this description, such as Real Time Strategy (RTS) or puzzle games, but they are outside the scope of our study.

Almost all commercial virtual reality games are based on a client-server architecture where a single server maintains the state of the game world or disjoint portion of the game world. The game state is typically structured as a collection of objects, each of which represents a part of the game world, such as the game world's terrain, players' avatars, computer controlled players (i.e., bots), items (e.g., health-packs), and projectiles. Each object is associated with a piece of code called a think

function that determines the actions of the object. Typical think functions examine and update the state of both the associated object and other objects in the game. For example, a monster may determine its move by examining the surrounding terrain and the position of nearby players. The game state and execution is composed from the combination of these objects and associated think functions.

The server runs a discrete event loop. In each iteration (or *frame* in game parlance), the server invokes think function for each object in the game and sends out the new view of the game state to each player. In FPS games, 10 to 20 iterations are executed each second; this frequency (called the *frame-rate*) is generally lower in other genres.

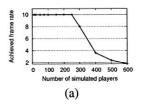
2.2 Client-Server Scaling Properties

The single server hosting a game can become a computation and communication bottleneck. To quantify these bottlenecks, we describe the general scaling properties of games and present measurements from Quake II, a typical FPS game.

Scalability Analysis: A game server's outbound bandwidth requirement (which is substantially higher than its inbound traffic [10]) is mainly determined by three game parameters: number of objects in play n, the average size of those objects s, and the game's frame-rate f. For example, in Quake II, if we only consider objects representing players (which tend to dominate game update traffic), n ranges from 8 to 64, s is about 200 bytes, and f is 10 updates per second. A naïve server implementation which simply broadcasts the updates of all objects to all c game clients would incur an outbound bandwidth cost of $c \cdot n \cdot s \cdot f$, or 1-66Mbps in games with 8 to 64 players.

Two common optimizations are employed to reduce this cost: area-of-interest filtering and delta-encoding. Since individual players only interact with a small portion of the game world at any given time, only updates about relevant objects are sent to the clients. Additionally, object state changes little from one update to the next. Therefore, most servers send the difference (i.e., delta) between updates. These optimizations reduce n and s respectively. In the case of 8-64 player Quake II games, the server bandwidth requirement reduces to about 62-492 kbps.

Empirical Scaling Behavior: Figure 1 shows the performance of a Quake II server running on a Pentium-III 1GHz machine with 512 RAM with different numbers of players. Each player is simulated using a server-side AI bot (though the server sends packets to them as if they were real clients). The server implements area-of-interest filtering, delta-encoding and does not rate-limit



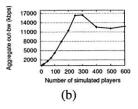
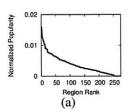


Figure 1: Computational and network load scaling behavior of a Quake II server.



Mean	14.8
Std Dev	3.6
Median	15
95 %ile	20

Figure 2: Quake III workload characteristics. (a) Popularity of different regions in a map. (b) Lengths of paths traveled by players every 10 seconds.

clients. Each game was run for 10 minutes at 10 frames per second.

As the computational load on a server increases, the server may require more than 1 frame-time of computation to service all clients. Hence, it may not be able to sustain the target frame-rate. Figure 1(a) shows the mean number of frames per second actually computed by the server, while Figure 1(b) shows the bandwidth consumed at the server for sending updates to clients. We note several points: First, as the number of players increases, area-of-interest filtering computation becomes a bottleneck and the frame-rate drops. (Detailed measurements show that the computational bottleneck is indeed the filtering code and not our AI bot code.) Second, Figure 1(b) shows that, as the number of players increases, the bandwidth-demand at the server increases more than linearly, since as the number of players increases, player interaction increases (for example, more missiles are fired.) Thus, n increases along with c resulting in a super-linear increase in bandwidth. Finally, we note that when the number of players exceeds 250, computational load becomes the bottleneck. The reduction in frame-rate offsets the increase in per-frame bandwidth (due to the increase in the number of clients), so we actually see the bandwidth requirement decrease. Although the absolute limits can be raised by employing a more powerful server, this illustrates that it is difficult for any centralized server to handle thousands of players.

2.3 A Multiplayer Gaming Workload

To further understand the requirements of online games, we studied the behavior of human players in real games. We obtained player movement traces from several ac-

Game	Servers
HalfLife /	10,582
Counter-Strike	
Quake 2	645
Quake 3	112
Tribes	233
Tribes 2	394

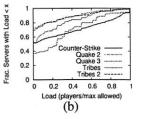


Figure 3: (a) Observed number of third-party deployed servers for several games, (b) Load on these servers.

tual Quake III games played by actual players on an Internet server. We observed that players tended to move between popular "waypoint" regions in the map and the popularity distribution of waypoints was Zipf-like. Figure 2(a) ranks the regions in a particular map by popularity (i.e., how often players occupy them.) This characteristic suggests that load balancing would be an important property of a distributed gaming architecture. Figure 2(b) shows the length of player movement paths in 10 second intervals, given in bucketized map units (the map is 20 units in diameter.) Despite the popularity of certain regions, players still move around aggressively in short periods of time; the median path length is 15, which is almost the diameter of the map. Hence, a distributed game architecture must be able to adapt to changes in player positions quickly.

Our analysis showed that this model fits the gameplay across several different maps and game types (e.g., Death Match and Capture the Flag), and we believe that it is representative of other FPS games since objectives and game-play do not vary substantially. Colyseus is designed primarily with FPS games in mind because we believe FPS game-play is the most difficult to support in a distributed setting. However, we discuss (Section 7.3) how games with different workloads might change our results.

2.4 Distributed Deployment Opportunities

Research designs [27], middle-ware layers [5, 17] and some commercial games [24] have used server clusters to improve the scaling of server-oriented designs. While this approach is attractive for publishers requiring tight administrative control, a widely distributed game deployment can address the scaling challenges and eliminate possible failure modes. In addition, a distributed design can make use of existing third party federated server deployments that we describe below, which is a significant advantage for small publishers.

There is significant evidence that given appropriate incentives, players are willing to provide resources for multiplayer games. For example, most FPS games servers are run by third parties – e.g., "clan" organizations formed by players. Figure 3(a) shows the number

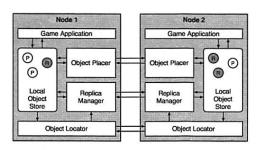


Figure 4: Colyseus components: Circled R's represent secondary replicas, circled P's represent primary objects.

of active third-party servers we observed for several different games. Figure 3(b) plots the cumulative distribution of load on the different sets of servers, where load is defined as the ratio of the number of active players on the server and the maximum allowed on the server. For most games, more than 50% of the servers have a load of 0. The server count and utilization suggest that there are significant resources that a distributed game design may use. Nonetheless, such a widely distributed deployment must address unique problems, such as inter-node communication costs and latencies.

3 Colyseus Architecture

Now, we present an overview of Colyseus, which primarily acts as a game object manager. There are two types of game objects: immutable and mutable. We assume that immutable objects (e.g., map geometry, game code, and graphics) are globally replicated (i.e., every node in the system has a copy) since they are updated very infrequently, if at all. Per-node storage requirements for Quake II and Quake III are about 500MB, though the vast majority of data is for graphics content, which could be elided on game servers. Colyseus manages the collection of mutable objects (e.g., players' avatars, computer controlled characters, doors, items), which we call the *global object store*.

Our architecture is an extension of existing game designs described in Section 2.1. In order to adapt them for a distributed setting, mutable objects and associated think functions are divided amongst participating nodes. Instead of running a single synchronous execution loop, in Colyseus, nodes run separate execution loops in parallel. Figure 4 shows the components in Colyseus that manage objects on each node, which we detail below.

State Partitioning: Each object in the global object store has a *primary* (authoritative) copy that resides on exactly one node. Updates to an object performed on any node in the system are transmitted to the primary owner, which provides a serialization order to updates. In addition to the primary copy, each node in the system may create a secondary replica (or *replica*, for short). These replicas enable remote nodes to execute code that ac-

cesses the object. Replicas are weakly consistent and are synchronized with the primary in an application dependent manner. In practice, the node holding the primary can synchronize replicas the same way viewable objects are synchronized on game clients in client-server architectures. Section 5 details the synchronization process and the consistency it affords game applications.

In summary, each node has a *local object store* which is a collection of primaries and replicas, a *replica manager* that synchronizes primary and secondary replicas, and a *object placer* which decides where to place and migrate primary replicas. For the purposes of this paper, we assume that objects are placed on the nodes closest to their controlling players, which is likely optimal for minimizing interactive latency, and defer details of the object placer and more sophisticated placement strategies to future work.

Execution Partitioning: Recall that existing games execute a discrete event loop that calls the think function of each object in the game once per frame. Colyseus retains the same basic design, except for one essential difference: a node only executes the think functions associated with *primary objects* in its local object store.

Although a think function could access any object in the game world, rarely will one require access to all objects simultaneously to execute correctly. Nonetheless, the execution of a think function may require access to objects that a node is not the primary owner of. In order to facilitate the correct execution of this code, a node must create secondary replicas of required objects. Fetching these replicas on-demand could result in a stall in game execution, violating real-time gameplay deadlines. Instead, each primary object predicts the set of objects that it expects to read or write in the near future, and Colyseus pre-fetches replicas of these objects. This prediction is specified as a selective filter on object attributes, which we call an object's area-ofinterest. We believe that most games can succinctly express their areas-of-interest using range predicates over multiple object attributes, which work especially well for describing spatial regions in the game world. For example, a player's interest in all objects in the visible area around its avatar can be expressed as a range query (e.g., $10 < x < 50 \land 30 < y < 100$). As a result, Colyseus maintains replicas that are within the union of its primaries' areas-of-interest in each node's local object store.

Object Location: Colyseus can use either a traditional randomized DHT or a *range-queriable* DHT as its *object locator*. Range-queries describing area-of-interests, which we call *subscriptions*, are sent and stored in the DHT. Other objects periodically publish metadata containing the current values of their *naming* attributes, such as their x, y and z coordinates, in the DHT. We call these

messages *publications*. A subscription and its matching publications are routed to the same location in the DHT, allowing the *rendezvous* node at which they meet to send all publications to their interested subscribers. Since nodes join the object location substrate in a fully self-organizing fashion, so there is no centralized coordination or dedicated infrastructure required in Colyseus.

A particular challenge in applying a DHT to object location in a real time setting is overcoming the delay between the submission of a subscription and the reception of matching publications. Section 6 details two methods to hide object location delays from the game application, and describes the trade-off between locality, dynamics, and complexity when using either DHT substrate in the context of locating game objects.

Application Interface: From our experience modifying Quake II to use Colyseus (described in Section 7) and our examinations of the source code of several other games, we believe that this model is sufficient for implementing most important game operations. Figure 5 shows the primary methods of interface for game objects managed by Colyseus. There are only two major additions to the centralized game programming model, neither of which is likely to be a burden on developers. First, each object uses GetLocation() to publish a small number of naming attributes. Second, each object specifies its area-of-interest in GetInterest() using range queries on naming attributes (i.e., a declarative variant of how area-of-interest is currently computed). A few additional interface methods exist for optimizations and are described in subsequent sections.

This architecture does not address some game components, such as content distribution (e.g., game patch distribution) and persistent storage (e.g., storing persistent player accounts). However, the problem of distributing these components is orthogonal to distributing gameplay and is readily addressed by other research initiatives [8, 9].

4 Evaluating Design Decisions

In order to evaluate design decisions in Colyseus, we developed our own distributed game based on the characteristics observed in Section 2.3. This section describes this initial workload and the experimental setup of micro-benchmarks we use in the subsequent sections to illustrate important aspects of Colyseus' design. In Section 7, we apply Colyseus in a distributed version of Quake II, demonstrating that our observations apply to an existing game.

4.1 Model Workload

We derived a *model workload* from our observations in Quake III games (see Section 2.3), which we im-

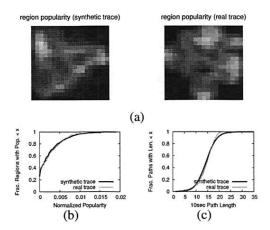


Figure 6: Comparison of our synthetic game trace with a real Quake III trace measured with human players.

plemented as a real game played by bots that runs on top of Colyseus. The game uses synthetic maps with randomly generated obstacles and bots move using a obstacle-sensitive mobility model based on Voronoi diagrams [18]. Mobility parameters like the probability of entering fights and staying at or leaving waypoints were based on trace values. In addition, area of interests are based on median interest sizes observed in Quake II and Quake III maps. Game mechanics such as object velocity, map size, and fight logic were based directly on values from Quake II and Quake III.

Figure 6 compares a trace based on our model workload with a real Quake III trace on a similar map. Part (a) shows the relative popularity of different regions in each map (lighter regions are more popular), where popularity is defined as how often players enter a given region. Although the maps are clearly different, we see that they share similar characteristics, such as several highly popular areas and less popular paths that connect them. Part (b) and (c) compare the distribution of region popularities and lengths of paths (in the number of regions) taken by players/bots during 10 second intervals, respectively. The distributions match up quite closely. Tan, et al. [29] concurrently developed a similar FPS mobility model (without fight logic) and found that it predicted client bandwidth and interest management accuracy well.

4.2 Experimental Setup

We emulate the network environment by running several virtual servers on 5-50 physical machines on Emulab [31]. The environment does not constrain link capacity, but emulates end-to-end latencies (by delaying packets) using measured pairwise Internet latencies sampled from the MIT King dataset [21]. Median round trip latencies for samples are between 80ms-90ms. Due to limited resources and to avoid kernel scheduling artifacts, when running several virtual servers on the same

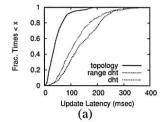
class ColyseusObject	
<pre>GetInterest(Interest* interest)</pre>	Obtain description of object's interests (e.g., visible area bounding box)
GetLocation(Location* locInfo)	Obtain concise description of object's location
GetVelocity(Vector* dir)	Obtain object's current velocity
IsInterested (ColyseusObject* other)	Decide whether this object is interested in another
PackUpdate(Packet* packet, BitMask mask)	Marshall update of object; bitmask specifies dirty fields for delta-encoding
UnpackUpdate(Packet* packet, BitMask mask)	Unmarshall an update for this object

Figure 5: The interface that game objects implement in applications running on Colyseus.

physical machine, we artificially dilate time (e.g., using a dilation factor of 3, 1 experimental minute lasts 3 actual minutes) by dilating all inter-node latencies, timers, and timeouts accordingly. Hence, our latency results do not include computational delays, but since our configurations emulated at most 8 players per server, computational delay would be negligible even in a real game (e.g., see Figure 1(a)). In addition, UDP is used for transport, so the impact time dilation would have on TCP does not affect our results. Each game/experiment run lasts 8 minutes, which is about half the time of a typical FPS game round.

Different experiments vary two main parameters: players-per-node and map-type. We use two player-pernode counts: 1 player per node, which we call the peerto-peer scenario (p2p), and 8 players per node, which we call the federated server scenario (fed). We use the p2p scenario to illustrate scaling behavior since it allows us to run the most virtual nodes per physical node in our testbed. Similarly, we use the fed scenario when quantifying the characteristics of a particular configuration, since it allows us to run the most total players in the game world, increasing interactivity. In general, increasing the number of players per node (while average density remains constant) increases communication costs linearly (since all players are randomly spawned in the map) and does not substantially affect the other metrics we measure (which are mostly functions of node count). We have validated these properties in most of our experiments.

We evaluate two types of maps: square (sqr) and rectangular (rect). In both types, we select the map area that achieves the same average player density as in a full Quake III game although the density distribution follows the Zipf-like model we observed. The height of rect maps is always equal to the diameter of a 16 player Quake III map, while sqr maps vary both dimensions equally. rect maps simulate a linearization (e.g., using Hilbert space-filling curves [26]) of a multi-dimensional map, which may be useful in some games where not much locality is sacrificed. Note that although the maps we use are uniform shapes, the area that is traversed during game-play obeys actual non-uniform characteristics, as demonstrated by Figure 6(a). The map type primarily impacts the performance of the object location layer,



Player	98.97
Missile	64.80
All	91.69

Figure 7: (a) Comparison of update latencies when sent directly through the topology and through a DHT. (b) Percentage of object updates that can bypass object location.

because the regions of each type will have different locality properties when mapped onto the DHT identifier space.

In the following sections, we describe the details of the replica manager and object locator, using the above setup to quantify important points. Experiments are named using the convention <code>node_count-{p2p,fed}-{sqr,rect}</code> to indicate their configurations.

5 Replica Management

The replica management component manages replica synchronization, responds to requests to replicate primaries on other nodes, and deletes replicas that are no longer needed. In our current implementation, primaries synchronize replicas in an identical fashion to how dedicated game servers synchronize clients: each frame, if the primary object is modified, a delta-encoded update is shipped to all replicas. Similarly, when a secondary replica is modified, a delta-encoded update is shipped to the primary for serialization. Although other update models are possible for games on Colyseus, this model is simple and reflects the same loose consistency in existing client-server architectures.

Decoupling Location and Synchronization: An important aspect of Colyseus' replica manager is the decoupling of object discovery and replica synchronization. Once a node discovers a replica it is interested in, it synchronizes the replica directly with the primary from that point on. The node periodically registers interest with the node hosting the primary to keep receiving updates to the replica.

Another strategy would be to always place each ob-

	roactive Rep an % Missir		es
Nodes	Players	On	Off
28	224	27.5	72.9
50	400	23.9	64.5
96	768	27.2	72.9

Table 1: Impact of proactive replication on missile object inconsistency.

ject on the node responsible for its region (as in *cell*-based architectures [17, 20, 24]). However, FPS game workloads exhibit rapid player movement between cells, which entails migration between servers. For example, in a 96-fed-rect game with one region per server, this approach causes each player to migrate once every 10 seconds, on average, and hence requires a frequency of connection hand-offs that would be disruptive to gameplay. Yet another design would be to route updates to interested parties via the rendezvous node in the DHT (as in [20]). However, this approach adds at least one extra hop for each update.

To quantify the impact of decoupling, Figure 7(a) compares the one-way direct latencies between 96 nodes in a real world end-host topology [21] (topology) and the delivery latency of publications and subscriptions in a 96-fed-rect experiment using both a range-queriable DHT (rangedht) and a traditional DHT (dht). Although routing through either substrate achieves much better than $\log n$ hops due to the effectiveness of route caching with a highly localized workload, the delays are still significantly worse than sending updates directly point-topoint, especially considering the target latency of 50-100ms in FPS games [1].

In Colyseus, the only time a node incurs the DHT latency is when it must discover an object which it does not have a replica of. This occurs when the primary just enters the area-of-interest of a remote object. Figure 7(b) quantifies how often this happens in the same game if each player were on a different node (the worst case). For each object type, the table shows the percentage of updates to objects that were previously in a primary's area-of-interest (and hence would already be discovered and not have to incur the lookup latency), as opposed to objects that just entered. For player objects almost 99% of all updates can be sent to replicas directly. For missiles, the percentage is lower since they are created dynamically and exist only for a few seconds, but over half the time missile replicas can still be synchronized directly also. Moreover, more aggressive interest prediction, which we discuss in the next section, would further increase the number of updates that do not need to be preceded by a DHT lookup, since nodes essentially discover objects before they actually need them.

Proactive Replication: To locate short-lived objects like missiles faster, Colyseus leverages the observation

that most objects originate at locations close to their creator, so nodes interested in the creator will probably be interested in the new objects. For example, a missile originates in the same location as the player that shot it. Colyseus allows an object to *attach* itself to others (via an optional AttachTo() method that adds to the object API in Figure 5). Any node interested in the latter will automatically replicate the former, circumventing the discovery phase altogether.

Table 1 shows the impact of proactive replication on the fraction of missiles missing (i.e., missiles which were in a primary's object store but not yet replicated) from each nodes' local object store (averaged across all time instances). We see that in practice, this simple addition improves consistency of missiles significantly. For example, in a 400 player game, enabling proactive replication reduces the average fraction of missiles missing from 64% to 24%. If we examined the object stores' 100ms after the creation of a missile, only 3.4% are missing on average (compared to 28% without proactive replication). The remainder of the missing missiles are more likely to be at the periphery of objects' areaof-interests and are more likely to tolerate the extra time for discovery. In addition, we note that the overhead is negligible.

Replica Consistency: In Colyseus, writes to replicas are tentative and are sent to the primary for serialization. Our model game applies tentative writes (tentatively), but a different game may choose to wait for the primary to apply it. In other words, individual objects follow a simple primary-backup model with optimistic consistency. The backup replica state trails the primary by a small time window ($\frac{1}{2}$ RTT, or, from the results shown in Figure 7(a), <100ms for 93% of node pairs), and are *eventually consistent* after this time window.

In addition to per-object consistency, it is desirable to consider view consistency in the context of a game. The view of a server (or a player) is the collection of objects that are currently within the union of the server's (player's) subscriptions. Here, we discuss view consistency with respect to the TACT model [32], since its continuous range of consistency/performance trade-offs likely to be most useful to game applications. In the TACT model, the view of a server can define a conit, or unit of consistency. There are two types of view inconsistency in Colyseus: first, a server is missing replicas for objects that are within its view; and second, replicas that are within its view are missing updates or have updates applied out-of-order. Both types of inconsistency actually exist in any application using the TACT model, since when a new conit is defined, time is required to first replicate the desired parts of the database to "initialize" the conit (resulting in the first type) before maintaining it (which can result in the second type). The first type is simply exacerbated in a distributed game because views change frequently and reads often can not wait for views to finish forming.

Since Colyseus introduces missing replicas as a significant source of inconsistency, we use the number of missing replicas as the primary metric when evaluating consistency. Inconsistency due to missing or late updates can be managed in an application specific manner using the TACT model (with game specified bounds on order, numerical, and staleness error). Hence, Colyseus is flexible enough to support games with different view consistency requirements.

We believe that most fast-paced games would rather endure temporary inconsistency rather than have the affects of writes (i.e., player actions) delayed, so our implementation adopts an optimistic consistency model with no bounds on order or numerical error in order to limit staleness as much as possible. As described above, this ensures replica staleness remains below 100ms almost all of the time. Limited staleness is usually tolerable in games since there is a fundamental limit to human perception at short time-scales and game clients can extrapolate or interpolate object changes to present players with a smooth view of the game [1]. Moreover, we observed that frequently occurring conflicts can be resolved transparently. For example, in our distributed Quake II implementation, the only frequent conflict that affects game-play is a failure to detect collisions between solid object on different nodes, which we resolve using a simple "move-backward" conflict resolution strategy when two objects are "stuck together." The game application can detect and resolve these conflicts before executing each frame.

6 Locating Distributed Objects

To locate objects, Colyseus implements a distributed location service on a DHT. Unlike other publish-subscribe services built on DHTs [6], the object locator in Colyseus must be able to locate objects using range queries rather than exact matches. Moreover, data items (i.e., object location information) change frequently and answers to queries must be delivered quickly to avoid degrading the consistency of views on different nodes in the system. In this section we describe three aspects of the object locator that enables it to meet these challenges. In addition, we describe how Colyseus can leverage *range-queriable* DHTs in its object locator design.

6.1 Location Overview

DHTs [28, 25] enable scalable metadata storage and location on a large number of nodes, usually providing a logarithmic bound on the number of hops lookups must traverse. With a traditional DHT, the object loca-

tor bucketizes the map into a discrete number of regions and then stores each publication in the DHT under its (random) region key. Similarly, subscriptions are broken up into DHT lookups for each region overlayed by the range query. When each DHT lookup reaches the rendezvous node storing metadata for that region, it returns the publications which match the original query back to the original node.

Range-queriable DHTs [3, 19] may be better fit to a distributed game architecture. Unlike traditional DHTs which store publications under discrete random keys to achieve load balance, a range-queriable DHT organizes nodes in a circular overlay where adjacent nodes are responsible for a contiguous range of keys. A range query is typically routed by delivering it to the node responsible for leftmost value in the range. This node then forwards the query to other nodes in the range. For example, using a range-queriable DHT, the object placer could use the x dimension attribute directly as the key. Since key values are stored continuously on the overlay (instead of randomly), range queries can be expressed directly, instead of having to be broken up into multiple DHT lookups. Moreover, object location metadata and queries are likely to exhibit spatial locality, which maps directly onto the overlay, allowing the object locator to circumvent routing paths and deliver messages directly to the rendezvous by caching recent routes. Finally, since nodes balance load dynamically to match the publication and subscription distribution, they may be able better handle the Zipf-like region popularity distribution observed in Section 2.

Colyseus implements both object location mechanisms, and we evaluate the trade-offs of each in Section 6.3.

6.2 Reducing Discovery Latency

Regardless of the underlying DHT substrate, the object locator in Colyseus provides two important primitives to reduce the impact of object discovery latency and overhead.

Interest Prediction and Aggregation: Spatial and temporal locality in object movement enables prediction of subscriptions (e.g., if an object can estimate where it will be in the near future, it can simply subscribe to that entire region as well). Colyseus expands a the bounding volume subscribed to by an object (via GetInterest()) using the following formula:

 $\begin{aligned} & Vol.min-= & PredTime \times PredMoveUpLeft + PubTime \\ & Vol.max+= & PredTime \times PredMoveDownRight + PubTime \end{aligned}$

This formula predicts the amount of movement an object will make in each direction per game time unit and multiplies it by the desired prediction time

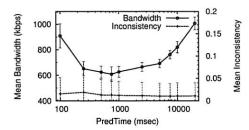


Figure 8: The impact of varying PredTime on total mean node bandwidth and local object store inconsistency.

(PredTime), which a per-object configuration parameter. The default implementation uses moving average of an object's velocity to estimate PredMoveUpLeft and PredMoveDownRight, but the application can override it (via an additional object API method) if more is known about an object's physics (e.g., missiles always move in a straight line). A small factor (PubTime) is added to account for the discovery and delivery time of publications for objects entering the object's subscription volume. Thus, if predicted subscriptions are stored in the DHT with a TTL = PredTime, it is unlikely they will have to be refreshed within that time. Subscription prediction amounts to *speculative pre-fetching* of object location attributes.

Speculation can incur overhead. Figure 8(a) shows the impact of tuning subscription prediction (by varying PredTime) in a 50-fed-rect game. The top line plots the total mean bandwidth required by each node, while the bottom line shows the mean local object store inconsistency, defined as the average fraction of missing player replicas in each node's object store across all time instances (an object is missing if it enters a primary's area-of-interest, but is not yet discovered). Error bars indicate one standard deviation.

The variation in bandwidth cost as we increase PredTime demonstrates the effects of speculation. When speculation time is too short (e.g., we only predict 100 ms or 1 frame into the future), each object must update subscriptions in the system more frequently, incurring a high overhead. If speculation time is too long, although objects can leave their subscriptions in the system for longer periods of time without updates, they receive a large number of false matches (publications which are in the speculated area-of-interest but not in the actual area-of-interest), also incurring overhead. Extraneous delivery of matched publications does not result in unnecessary replication, since upon reception of a prefetched publication, a node will cache (for the length of the TTL) and periodically check whether it actually desires the publishing object by comparing the publication to its up-to-date unpredicted subscriptions locally. Hence, overhead is solely due to extra received publications. In this particular configuration, the "sweet-spot"

is setting PredTime around 1 second. Although this optimal point will vary depending on game characteristics (e.g., density, update size, etc.), notice that we are able to maintain the same level of inconsistency regardless of the PredTime value. Hence, we can automatically optimize PredTime without affecting the level of inconsistency observed by the game. In addition, although we focused on using prediction to minimize communication overhead, we can also trade-off overhead for improved consistency by increasing PubTime.

To further reduce subscription overhead, Colyseus enables aggregation of overlapping subscriptions using a local *subscription cache*, which recalls subscriptions whose TTLs have not yet expired (and, thus, are still registered in the DHT), and an optional *aggregation filter*, which takes multiple subscriptions and merges them if they contain sufficient overlap. This filter uses efficient multi-dimensional box grouping techniques originally used in spatial databases [15].

Soft State Storage: In most publish-subscribe systems, only subscriptions are registered and maintained in the DHT while publications are not. The object locator stores both publications and subscriptions as soft state at the rendezvous, which expire them after a TTL carried by each item. When a subscription arrives, it matches with all currently stored publications, in addition to publications that arrive after it.

This design achieves two goals: First, if only subscriptions were stored, subscribers would have to wait until the next publication of an interesting object before it would be matched at the rendezvous. By storing publications, a subscription can immediately be matched to recent publications. This suffices for informing the node about relevant objects due to spatial locality of object updates. Second, different types of objects change their naming attributes at different frequencies (e.g., items only change locations if picked up by a player), so it would be wasteful to publish them all at the same rate. Moreover, even objects with frequently changing naming attributes can publish at lower rates (with longer TTLs) by having subscription prediction take into account the amount of possible staleness (i.e., we add PubTTL×Velocity to the PubTime factor above, accounting for how far an object could have moved between publication intervals).

6.3 Comparison of Routing Substrates

In this section, we evaluate how the performance of Colyseus is affected by the choice of the routing substrate: a traditional DHT (dht) versus a range-queriable DHT (rangedht). In general, our results show rangedht incurs lower bandwidth overhead compared to a dht by utilizing contiguity in data placement and has good scaling properties if the game map can be linearized.

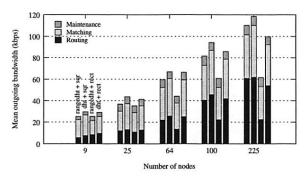


Figure 9: Scaling of per-node bandwidth using a dht and rangedht in a p2p game with sqr and rect maps.

rangedht incurs higher object discovery latency compared to a dht, but at time scales of 100ms, the resultant inconsistency in game-state is indistinguishable. Finally, rangedht more fairly balances object location overhead between all nodes, suggesting that it is more suitable in bandwidth constrained deployments.

Colyseus uses an implementation of Mercury [3] with the extensions described earlier in this section. Mercury is used both as the dht and the rangedht, handling publications and subscriptions as described in Section 6.1. When used as a dht, Mercury breaks up each map into a number of regions equal to the number of players in a map. When used as a rangedht, the \times dimension is used as the key attribute. In both cases, each node caches $2\log(n)$ recently used routes.

6.3.1 Communication Costs

Figure 9 compares the average per-node outbound bandwidth requirements for object discovery, varying the number of nodes and the map type in p2p games. The bandwidth value reported by each node is the mean taken over a 5-minute period in the middle of the experiment. Bandwidth is divided into three components: sending and routing publications and subscriptions in Mercury (routing), delivering matched publications and subscriptions (matching), and DHT maintenance (maintenance). In all cases, rangedht consumes less bandwidth than dht.

Performance of a dht is similar for both sqr and rect maps. However, a rangedht performs noticeably better with rect maps because the total span of the key-space is larger relative to the width of subscriptions, so each subscription covers fewer nodes.

Scaling Behavior: Since the map area grows linearly with the number of players and subscription area is constant, as more nodes are added to a rangedht, the number of nodes contacted for each subscription stays constant if using a rect map, but grows proportional to \sqrt{n} if using a sqr map. For a dht substrate, this number stays constant irrespective of the map type. However, the lack of locality in the generated subscriptions results in higher rout-

Metric		dht	loadbal
Per-node total bwidth	std-dev	0.30	0.15
(normalized by mean)	max	1.93	1.45
Per-node matching bwidth	std-dev	1.01	0.57
(normalized by mean)	max	4.41	2.57
Avg. % missing replicas		8%±6%	10%±9%

Table 2: Effectiveness of a load-balanced rangedht. The percentage of missing replicas shows the mean and standard deviation.

ing overhead since caching routes becomes less effective. In addition to these effects, since player interaction grows as the number of players in the game increase, the overall matching traffic also grows (as Figure 9 shows). Hence, we observe that both dht and rangedht routing bandwidth scale poorly using sqr maps, but rangedht scales well with a linearized rect map.

Load Balancing: Since popularity of the regions in the model workload is Zipfian, nodes in the routing ring responsible for such regions can get considerably more traffic than others. We now focus on the effectiveness of the leave-join load-balancing mechanisms built into the Mercury rangedht, which dynamically moves lightly loaded nodes to heavily loaded regions the DHT. The number of publications and subscriptions routed per second, averaged over a 30-second moving window, is used as the measure of the load.

Table 2 compares the bandwidth and view inconsistency (see Section 6.3.2) for a 96-fed-rect game. We find that a rangedht with load balancing enabled (loadbal) reduces the maximum per-node bandwidth by about 25% (relative to the mean) and the maximum per-node matching bandwidth by about 42%, compared to a dht. While partitioning a busy range may not necessarily result in decreasing routing load (since each subscription will have to visit all nodes that span its range), it is effective at partitioning the matching load which is a significant component of the total bandwidth costs (see Figure 9). Also, the average fraction of missing replicas is not substantially higher, suggesting that players do not lose many updates due to the leave-join dynamics of load balancing.

6.3.2 Latency and Inconsistency

In this section, we evaluate the impact of the routing substrate on game-state consistency. We first evaluate how long it takes for a node to discover and replicate an object that it is interested in, which provides an estimate of the worst case delay that a view might have to endure. We then examine the impact that this latency has on the consistency of local object stores on different nodes.

Discovery Latency: Figure 10 shows the median time elapsed between submitting a subscription and generation of a matching publication for the different DHTs and map types. This latency is broken down into two

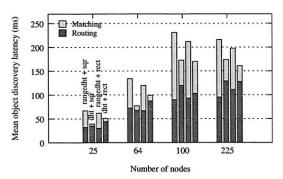


Figure 10: The mean time required to discover replica of an object once a subscription is generated.

parts: routing the subscription to the *first* (left-most) rendezvous (routing), and delay incurred at the rendezvous before a matching publication arrives (matching). To completely construct a replica, an additional delay of 1.5 RTT (135ms on average) must be added: 0.5 RTT for delivering the publication, and 1 RTT for fetching the replica. However, this latency is independent of the location substrate.

The routing delay for subscriptions scales similarly in both DHTs, as expected. Both are able to exploit caching so the median hop count is at most 3 in all cases. However, the matching latency is higher for the rangedht case. This is because the matching component incorporates the latency incurred when spreading the subscription *after* reaching the left-most rendezvous point. Thus, dht incurs bandwidth overhead by sending multiple disjoint subscriptions, but obtains an small overall latency advantage.

Discovery latencies are only incurred when an interesting object is first discovered (e.g., when a player enters a new room or an object enters the periphery of a player's visible area). Once a replica is discovered and created, it will be kept up to date through direct communication with the primary. Hence replica staleness will be tied to the latency distribution of the topology, which is less than 100ms for most node pairs (see Section 5). Incorporating proximity routing techniques [14] into our Mercury implementation can further reduce the latency of the routing component in both cases.

View Inconsistency: Now we examine the impact of discovery delay on view consistency. We define view consistency as the ratio of replicas missing and total replicas in a node's subscriptions (summed over all game frames). Figure 11 shows the fraction of replicas missing for a dht and rangedht in p2p-sqr games, if we allow 0ms, 100ms, and 400ms to elapse after the objects enter a node's subscriptions.

We see that inconsistency in game state is approximately the same irrespective of the choice of the routing substrate. The rangedht has slightly higher inconsis-

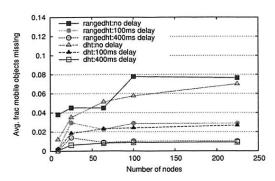


Figure 11: The fraction of replicas missing averaged across all time instances as we scale the number of servers.

tency due to the higher object discovery latency. However, this difference vanishes if we allow for a small delay of 100ms. For both DHTs, the inconsistency is fairly low. For example, with 64 nodes, about 4% of the objects required are missing at any given time. This improves to about 2% missing if we allow for a 100ms delay (1 frame), and it improves to 1% missing if we allow for a 400ms delay (4 frames).

7 Evaluation With a Real Game

To demonstrate the practicality of our system, we modified Quake II to use Colyseus. In our Quake II implementation, we represent an object's area-of-interest with a variable-sized bounding box encompassing the area visible to the object. We automatically delta-encode and serialize Quake II objects using field-wise diffs, so the average object delta size in our implementation is 145 bytes. Quake II's server to client messages are more carefully hand-optimized and average only 22 bytes. Unmodified Quake II clients can connect to our distributed servers and play the game with an interactive lag similar to that obtained with a centralized server. As a result, the system can be run as a peer-to-peer application (with every client running a copy of the distributed server) or as a distributed community of servers.

We use a large, custom map with computer controlled bots as the workload, and the same Emulab testbed setup described in section 4 for our Quake II evaluation. However, we did *not* artificially dilate time, so all numbers reported take into account actual execution times. We use the Mercury rangedht as the object location substrate, and linearize the game map when mapping it onto the DHT. Further details about our Quake II prototype and additional results can be found in an associated technical report [4].

7.1 Communication Cost

Figure 12 compares the bandwidth scaling of Colyseus running p2p games with the client-server and broadcast architecture alternatives. We simulate the alternatives

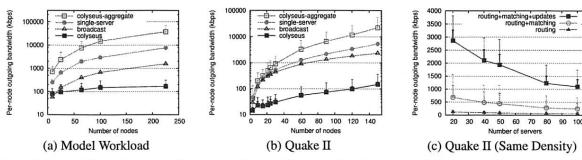


Figure 12: Bandwidth scaling properties using (a) the model workload and (b) the Quake II workload (note the logarithmic scale). Part (c) shows the scaling of Quake II, with a constant number of players.

using the same game-play events as the real execution on Colyseus.

Figure 12(a) shows the scaling properties with rect maps under our model workload. The workload keeps mean player density constant by increasing the map size. The thin error bar indicates the 95th percentile of 1 second burst rates across all nodes, while thick error bars indicate 1 standard deviation from the mean. The colvseus and broadcast lines show per-node bandwidth while the colyseus-aggregate line shows the total bandwidth used by all nodes in the system. At very small scales (e.g., 9 players), the overhead introduced by object location is high and Colyseus performs worse than broadcast. As the number of nodes increases, each node in Colyseus generates an order of magnitude less bandwidth than each broadcast node or a centralized server. Moreover, we see that Colyseus' per-node bandwidth costs rise much more slowly with the number of nodes increase than either of the alternatives. Nonetheless, the colyseus-aggregate line shows that we do incur an overall overhead factor of about 5. This is unlikely to be an issue for networks with sufficient capacity.

Figure 12(b) shows the same figure when running with the Quake II workload. We observe similar scaling characteristics here, except that the per-node Colyseus bandwidth appears to scale almost quadratically rather than less-than-linearly as in our model workload. This is primarily due to the fact that the Quake II experiments were run on the same map, regardless of the number of players. Thus, the average density of players increased with the number of nodes, which adds a quadratic scaling factor to all four lines. To account for this effect, Figure 12(c) shows how each component of Colyseus' traffic scales (per node) if we fixed the number of players in the map at 400 and increase the number of server nodes handling those players (by dividing them equally among the nodes). Due to inter-node interests between objects, increasing the number of nodes may not reduce per-node bandwidth cost by the same factor. In this experiment, we see a 3-fold decrease in communication cost per node with a 5-fold increase in the number of

nodes, so overhead is less than a factor of 2. We expect similar bandwidth scaling characteristics to hold for our model workload and Quake II if average player density were fixed. This result shows that the addition of resources in a federated deployment scenario can effectively reduce per-node costs.

If we hand-tune update delta sizes so they were smaller, the client-server and broadcast architectures would perform better. However, Figure 12(c) shows that updates also account for over 75% of Colyseus' costs, so Colyseus would get a substantial benefit as well. Moreover, the scaling properties would not change.

7.2 View Inconsistency

We now examine the view inconsistency, i.e., fraction of missing local replicas, observed in the Quake II workload (Section 6.3.2 showed this for the model workload.) Figure 13(a) shows the fraction of replicas missing as we scale the number of nodes for a p2p scenario. The results are very similar to those obtained with the model workload. Note that nearly one half of the replicas a node is missing at any given time instance arrive within 100ms and less than 1% take longer than 400ms to arrive.

Figure 13(b) shows the cumulative distribution of the number of missing objects for a 40-fed game. On average, a node requires 23 remote replicas at a given time instance. About 40% of the time, a node is missing no replicas; this improves to about 60% of the time if we wait 100ms for a replica to arrive and to over 80% of the time if we wait 400ms for a replica to arrive. The inconsistency is less for sparser game playouts.

Although the fraction of missing replicas is low, objects in a view can differ in semantic value; e.g., it is probably more important to promptly replicate a missile that is about to kill a player than a more distant object. In general, a game-specific inconsistency metric might consider the type, location, and state of missing objects to reflect the total impact on game-play quality. Due to locality in object movement, Colyseus' replication model accounts for at least one important aspect: location. Figure 13(c) compares the distance (over time)

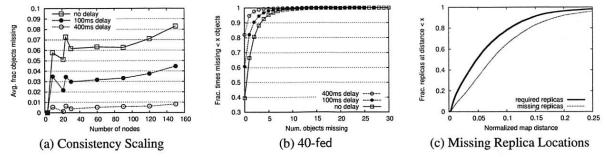


Figure 13: (a) Mean fraction of replicas missing as we vary the number of servers/players in Quake II. (b) CDF of missing objects in a 40-fed game. (c) CDF showing the distance of missing replicas from a subscriber's origin.

of a player to objects in its area-of-interest and the distance to those that are missing. Replicas that are missing from a view tend to be closer to the periphery of object subscriptions (and hence, farther away from the subscriber and probably less important). The difference in the distributions is not larger because subscription sizes in Quake II are variable, so objects at the periphery of a subscription may still be close to a player if they are in a small room. We leave a more game-play-centric evaluation of view inconsistency to future work.

7.3 Discussion

Throughout our evaluation of Colyseus we have used workloads derived from Quake II or Quake III, which we believe are representative of FPS games in general. However, questions remain about how representative our results are to other game genres, such as massively multiplayer Role Playing Games (RPGs.)

RPGs have lower update rates and have much smaller per-player bandwidth requirements than FPS games [7]. Hence, they are usually designed to tolerate much longer delays in processing player actions. In general, these characteristics imply that an RPG game implemented on Colyseus would incur lower communication costs than what we have measured. We do not expect discovery delay and replica staleness to change substantially because they are primarily functions of system size and network topology. Consistency may actually improve since players generally move slower in RPG games, and players have a higher tolerance for inconsistency (lower update rates imply existing game clients already tolerate staler state.) Thus, although we have demonstrated two case studies that effectively used Colyseus, we believe it can also be applied to less demanding game genres.

8 Related Work

There are a number of other commercial and research game architectures. Some games (e.g., MiMaze [12] and most Real Time Strategy (RTS) games [2]) use *parallel simulation*, where each player simulates the entire game world. All objects are globally replicated and kept

consistent using lock-step synchronization and update broadcast, resulting in quadratic scaling behavior and limiting response time to the speed of the slowest client. These deficiencies are tolerated in RTS games because they rarely involve more than 8 players.

Second-Life [24] and Butterfly.net [17] perform interest filtering by partitioning the game world into disjoint regions called *cells*. SimMUD [20] makes this approach fully distributed by assigning cells to keys in a DHT, though, unlike Colyseus, primaries in SimMUD reside on the rendezvous node. Although these approaches share some traits with Colyseus, we believe that we are the first to demonstrate the feasibility of implementing a real-world game on a distributed architecture that is (1) not designed for a centralized cluster ([24, 17]), and (2) that supports FPS games, which have much tighter latency constraints than RPGs (which were targeted by SimMUD). Furthermore, using a cell-based design with an FPS game can result in frequent object migration, as shown in Section 5.

Several architectures proposed for Distributed Virtual Reality environments and distributed simulation (notably, DIVE [11], MASSIVE [13], and High Level Architecture (HLA) [16]) have similar goals as Colyseus but focus on different design aspects. DIVE and MASSIVE focus on sharing audio and video streams between participants while HLA is designed for military simulations. None address the specific needs of modern multiplayer games and, to our knowledge, none have been demonstrated to scale to hundreds of participants without the use of IP multicast.

9 Summary and Future Work

This paper described the design, implementation and evaluation of a distributed architecture for online multiplayer games. Colyseus enables low-latency game-play via three important design choices: (1) decoupling object discovery and replica synchronization, (2) proactive replication for short-lived objects, and (3) pre-fetching of relevant objects using interest prediction. Our investigation showed that a range-queriable DHT achieves bet-

ter scalability and load balance than a traditional DHT when used as a object location substrate, with a small consistency penalty. We believe our adaptation of a commercial game (Quake II) demonstrates the practicality of Colyseus' design.

Nonetheless, our work on Colyseus is on-going. For example, Colyseus enables three new avenues for cheating: (1) nodes can modify objects in their local store in violation of game-play logic (2) nodes can withhold publications or updates of objects they own, and (3) nodes can subscribe to regions of the world that they should not "see." Although our work on addressing cheating is nascent, we believe we can leverage Colyseus' flexibility in object placement by carefully selecting the owners of primary objects to limit the damage inflicted by malicious nodes. Moreover, nodes holding replicas can act as witnesses to detect violations of game-play rules.

For more information about the project (software, documentation and announcements), please visit: http://www.cs.cmu.edu/~ashu/gamearch.html

10 Acknowledgements

We would like to thank our shepherd Alex Snoeren and the anonymous reviewers for their comments and suggestions. James Hayes and Sonia Chernova collected the Quake III traces we based our model game on. This work was funded by a grant from the Technology Collaborative.

References

- BEIGBEDER, T., ET AL. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003. In *NetGames* (Aug. 2004).
- [2] BETTNER, P., AND TERRANO, M. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *Gamasutra* (Mar. 2001).
- [3] BHARAMBE, A., ET AL. Mercury: Supporting scalable multi-attribute range queries. In SIGCOMM (Aug. 2004).
- [4] BHARAMBE, A., ET AL. A Distributed Architecture for Interactive Multiplayer Games. Tech. Rep. CMU-CS-05-112, CMU, Jan. 2005.
- [5] Big World. http://www.microforte.com.
- [6] CASTRO M., ET AL. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. on Sel. Areas in Comm. 20*, 8 (Oct. 2002).
- [7] CHEN, K., ET AL. Game Traffic Analysis: An MMORPG Perspective. In NOSSDAV (June 2005).
- [8] DABEK, F. ET AL. Wide-area cooperative storage with CFS. In SOSP (Oct. 2001).
- [9] DRUSCHEL, P., AND ROWSTRON, A. Storage management and caching in PAST, a large-scale, persistent peerto-peer storage utility. In SOSP (Oct. 2001).
- [10] FENG, W., ET AL. Provisioning on-line games: A traffic analysis of a busy counter-strike server. In *IMW* (Nov. 2002).

- [11] FRÉCON, E., AND STENIUS, M. DIVE: A scaleable network architecture for distributed virtual environments. *Dist. Sys. Eng. J.* 5, 3 (1998), 91–100.
- [12] GAUTIER, L., AND DIOT, C. MiMaze, A Multiuser Game on the Internet. Tech. Rep. RR-3248, INRIA, France, Sept. 1997.
- [13] GREENHALGH, C., ET AL. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS* (June 1995).
- [14] GUMMADI, K. P., ET AL. The Impact of DHT Routing Geometry on Resilience and Proximity. In SIGCOMM (Aug. 2003).
- [15] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In SIGMOD (June 1984).
- [16] IEEE standard for modeling and simulation high level architecture (HLA), Sept. 2000. IEEE Std 1516-2000.
- [17] IBM and Butterfly to run PlayStation 2 games on Grid. http://www-1.ibm.com/grid/announce_ 227.shtml, Feb. 2003.
- [18] JARDOSH, A. ET AL. Towards Realistic Mobility Models for Mobile Ad hoc Network. In MOBICOM (Sept. 2003).
- [19] KARGER, D., AND RUHL, M. Simple efficient loadbalancing algorithms for peer-to-peer systems. In *IPTPS* (Feb. 2004).
- [20] KNUTSSON, B. ET AL. Peer-to-peer support for massively multiplayer games. In *INFOCOM* (July 2004).
- [21] MIT King Data. http://www.pdos.lcs.mit. edu/p2psim/kingdata.
- [22] Quake II. http://www.idsoftware.com/ games/quake/quake2.
- [23] Quake III Arena. http://www.idsoftware.com/ games/quake/quake3-arena.
- [24] ROSEDALE, P., AND ONDREJKA, C. Enabling Player-Created Online Worlds with Grid Computing and Streaming. *Gamasutra* (Sept. 2003).
- [25] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Middleware* (Nov. 2001).
- [26] SAGAN, H. Space-Filling Curves. Springer-Verlag, New York, NY, 1994.
- [27] SHAIKH, A., ET AL. Implementation of a Service Platform for Online Games. In *NetGames* (Aug. 2004).
- [28] STOICA, I., ET AL. Chord: A scalable peer-to-peer lookup service for internet applications. In SIGCOMM (Aug. 2001).
- [29] TAN, S. A., ET AL. Networked game mobility model for first-person-shooter games. In *NetGames* (Oct. 2005).
- [30] Torque Networking Library. http://www.opentnl.org.
- [31] WHITE, B., ET AL. An integrated experimental environment for distributed systems and networks. In OSDI (Dec. 2002).
- [32] YU, H., AND VAHDAT, A. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. ACM Trans. on Comp. Sys. (Aug. 2002).

Na Kika: Secure Service Execution and Composition in an Open Edge-Side Computing Network

Robert Grimm, Guy Lichtman, Nikolaos Michalakis, Amos Elliston, Adam Kravetz, Jonathan Miller, and Sajid Raza New York University

Abstract

Making the internet's edge easily extensible fosters collaboration and innovation on web-based applications, but also raises the problem of how to secure the execution platform. This paper presents Na Kika, an edge-side computing network, that addresses this tension between extensibility and security; it safely opens the internet's edge to all content producers and consumers. First, Na Kika expresses services as scripts, which are selected through predicates on HTTP messages and composed with each other into a pipeline of content processing steps. Second, Na Kika isolates individual scripts from each other and, instead of enforcing inflexible a-priori quotas, limits resource consumption based on overall system congestion. Third, Na Kika expresses security policies through the same predicates as regular application functionality, with the result that policies are as easily extensible as hosted code and that enforcement is an integral aspect of content processing. Additionally, Na Kika leverages a structured overlay network to support cooperative caching and incremental deployment with low administrative overhead.

1 Introduction

Web-based applications increasingly rely on the dynamic creation and transformation of content [5]. Scaling such applications to large and often global audiences requires placing them close to clients, at the edge of the internet. Edge-side content management provides the CPU power and network bandwidth necessary to meet the needs of local clients. As a result, it reduces load on origin servers, bandwidth consumption across the internet, and latency for clients. It also absorbs load spikes, e.g., the Slashdot effect, for underprovisioned servers. Based on similar observations, commercial content distribution networks (CDNs) already offer edge-side hosting services. For example, Akamai hosts customer-supplied J2EE components on edge-side application servers [1]. Furthermore, many ISPs provide value-added services, such as "web accelerators", by dynamically transforming web content on the edge. However, commercial CDNs and ISPs have limited reach. To manage the trust necessary for exposing their hosting infrastructure to other people's code, they rely on traditional, contract-based business relationships. As a result, commercial CDNs

and ISPs are ill-suited to collaborative and communitybased development efforts; they best serve as amplifiers of (large) organizations' web servers.

At the same time, many community-based efforts are exploring the use of web-based collaboration to address large-scale societal and educational problems. For instance, researchers at several medical schools, including New York University's, are moving towards web-based education [10, 43, 45] to address nationally recognized problems in medical education [28, 49]. The basic idea is to organize content along narrative lines to re-establish context missing in clinical practice, complement textual presentation with movies and animations to better illustrate medical conditions and procedures, and leverage electronic annotations (post-it notes) and discussions for building a community of students and practitioners. Furthermore, such web-based educational environments dynamically adapt content to meet students' learning needs and transcode it to enable ubiquitous access, independent of devices and networks. A crucial challenge for these efforts is how to combine the content and services created by several groups and organizations into a seamless learning environment and then scale that environment to not only the 67,000 medical students in the U.S., but also the 850,000 physicians in the field as well as to medical personnel in other countries facing similar problems.

Taking a cue from peer-to-peer CDNs for static content, such as CoDeeN [47, 48] and Coral [13], Na Kika¹ targets cooperative efforts that do not (necessarily) have the organizational structure or financial resources to contract with a commercial CDN or cluster operator and seeks to provide an edge-side computing network that is fully open: Anyone can contribute nodes and bandwidth to Na Kika, host their applications on it, and access content through it. In other words, by opening up the internet's edge, Na Kika seeks to provide the technological basis for improved collaboration and innovation on large-scale web-based applications. In this paper, we explore how Na Kika addresses the central challenge raised by such an open architecture: how to secure our execution platform while also making it easily extensible.

Na Kika, similar to other CDNs, mediates all HTTP in-

¹Our system is named after the octopus god of the Gilbert Islands, who put his many arms to good use during the great earth construction project.

teractions between clients and servers through edge-side proxies. Also similar to other CDNs, individual edgeside nodes coordinate with each other to cache content, through a structured overlay in our case. Na Kika's key technical difference—and our primary contribution—is that both hosted applications and security policies are expressed as scripted event handlers, which are selected through predicates on HTTP messages and composed into a pipeline of content processing stages. Our architecture builds on the fact that HTTP messages contain considerable information about clients, servers, and content to expose the same high-level language for expressing functionality and policies alike-with the result that policies are as easily extensible as hosted code and that enforcement is an integral aspect of content processing. A second difference and contribution is that Na Kika's resource controls do not rely on a-priori quotas, which are too inflexible for an open system hosting arbitrary services with varying resource requirements. Instead Na Kika limits resource consumption based on congestion: If a node's resources are overutilized, our architecture first throttles requests proportionally to their resource consumption and eventually terminates the largest resource consumers.

Our use of scripting and overlay networks provides several important benefits. First, scripting provides a uniform and flexible mechanism for expressing application logic and security policies alike. Second, scripting simplifies the task of securing our edge-side computing network, as we can more easily control a small execution engine and a small number of carefully selected platform libraries than restricting a general-purpose computing platform [20, 41]. Third, scripting facilitates an API with low cognitive complexity: Na Kika's event-based API is not only easy to use but, more importantly, already familiar to programmers versed in web development. Fourth, the overlay ensures that Na Kika is incrementally scalable and deployable. In particular, the overlay supports the addition of nodes with minimal administrative overhead. It also helps with absorbing load spikes for individual sites, since one cached copy (of either static content or service code) is sufficient for avoiding origin server accesses.

At the same time, *Na Kika* does have limitations. Notably, it is unsuitable for applications that need to process large databases, as the databases need to be moved to the internet's edge as well. Furthermore, since *Na Kika* exposes all functionality as scripts, applications whose code needs to be secret cannot utilize it (though obfuscation can help). Next, by utilizing *Na Kika*, content producers gain capacity but also give up control over their sites' performance. We expect that any deployment of our edge-side computing network is regularly monitored to identify persistent overload conditions and to

rectify them by adding more nodes. Finally, while *Na Kika* protects against untrusted application code, it does trust edge-side nodes to correctly cache data and execute scripts. As a result, it is currently limited to deployments across organizations that can be trusted to properly administer local *Na Kika* nodes. We return to this issue in Section 6.

2 Related Work

Due to its palpable benefits, several projects have been exploring edge-side content management. A majority of these efforts, such as ACDN [33], ColTrES [8], Tuxedo [38], vMatrix [2], and IBM's WebSphere Edge Server [17] (which is used by Akamai), explore how to structure the edge-side hosting environment. Since they are targeted at closed and trusted deployments, they do not provide an extension model, nor do they include the security and resource controls necessary for hosting untrusted code. In contrast, the OPES architecture for edgeside services recognizes the need for extensibility and service composition [4, 23]. While it does not specify how composition should be achieved, OPES does define potential security threats [3]. Their scope and magnitude is illustrated by experiences with the CoDeeN open content distribution network [48].

Next, Active Cache [9] and SDT [19] enable content processing in proxy caches. While they do not provide an extension mechanism, they do provide precise control over edge-side processing through server-specified HTTP headers. Furthermore, while SDT enforces only coarse-grained resource controls for Perl and none for Java, Active Cache executes Java code with resource limits proportional to the size of the content being processed. Unlike these systems, Pai et al.'s proxy API [31] provides fine-grained extensibility for web proxies through an event-based API akin to ours. At the same time, their work focuses on enabling high-performance extensions in trusted deployments, while our work focuses on containing arbitrary extensions in untrusted deployments. Finally, Active Names [46] are explicitly designed for extensibility and service composition, chaining processing steps in a manner comparable to Na Kika's scripting pipeline. In fact, by introducing a new naming interface, Active Names offer more flexibility for content processing than our work. However, they also require a new service infrastructure, while Na Kika integrates with the existing web.

While cooperative caching has its limitations [50], coordination between edge-side nodes is still important for scaling a system, in particular to balance load and absorb load spikes. To this end, CoDeeN [47], ColTrES [8], and Tuxedo [38] are exploring the use of domain-specific topologies and algorithms. In contrast, *Na Kika* leverages previous work on structured overlay

networks [13, 16, 35, 42, 52] for coordinating between local caches. We believe that structured overlays provide a robust and scalable alternative to domain-specific coordination strategies. Additionally, structured overlays have already been used successfully for caching static content [13, 18].

In most edge-side systems, nodes cannot be entrusted with the sole copies of application data, and hard state requiring stronger consistency than the web's expirationbased guarantees (or lack thereof) must remain on origin servers. In contrast, ACDN [33] reduces access latency for such data by replicating it across edge nodes and by providing full serializability through a primary replica. Gao et al. [14] explore alternative replication strategies by exposing a set of distributed objects that make different trade-offs between consistency, performance, and availability. Alternatively, the continuous consistency model provides a framework for expressing such trade-offs through a uniform interface to hard state [51]. Na Kika's support for application state builds on Gao et al.'s approach, with the primary difference that replicated state is subject to Na Kika's security and resource controls.

Web content processing is (obviously) not limited to edge nodes and can be performed on servers and clients as well. For example, Na Kika has several similarities with the cluster-based TACC architecture [12]. Both Na Kika and TACC rely on a pipeline of programs that process web content, and both build on the expiration-based consistency model of the web to cache both original and processed content. Na Kika differs in that it targets proxies distributed across the wide area and and thus needs to carefully contain hosted code. Comparable to Na Kika, DotSlash [53] helps absorb load spikes by moving script execution to other servers in a "mutual-aid community". Unlike Na Kika, it has no extension model and does not provide security and resource controls. At the other end, client side includes [34] (CSI) move the assembly of dynamic content to the client, which can improve latency for clients relying on low bandwidth links. However, due to their focus on assembling content fragments, CSI are not suitable for content processing in general. The underlying edge side includes [29, 44] (ESI) can easily be supported within Na Kika.

Finally, based on the realization that system security can clearly benefit from a dedicated and concise specification of policies, a considerable number of efforts have explored policy specification languages. For example, domain and type enforcement [7], XACML [22], and trust management systems such as PolicyMaker, KeyNote, and SPKI [6, 11] include languages for expressing and enforcing policies. All these systems require explicitly programmed calls to the respective reference monitor. In contrast, previous work on security

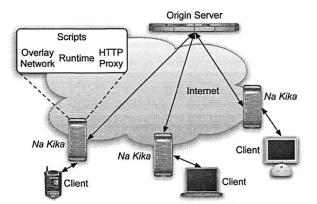


Figure 1: Illustration *Na Kika*'s architecture. Edge-side proxies mediate all HTTP interactions between clients and servers by executing scripts; proxies also coordinate with each other through an overlay network.

for extensible systems advocates the separation of policies, enforcement, *and* functionality and relies on binary interposition to inject access control operations into executing code [15, 39]. The WebGuard policy language relies on a similar approach for securing web-based applications [40]. Since *Na Kika*'s programming model is already based on interposition, we leverage the same predicate selection mechanism for application logic and policies, thus eliminating the need for a separate policy specification language.

3 Architecture

Like other extensions to the basic web infrastructure and as illustrated in Figure 1, Na Kika relies on proxies that mediate HTTP interactions between clients and servers. To utilize these proxies, content producers and consumers need to change existing web practices along two lines. First, content producers need to publish the necessary edge-side processing scripts on their web sites. Content producers need not provide scripts for an entire site at once. Rather, they can transition to Na Kika piecemeal, starting with content whose creation or transformation exerts the highest resource demands on their servers. Second, links need to be changed by appending ".nakika.net" to a URL's hostname, so that Na Kika's name servers can redirect clients to (nearby) edge nodes. As described in [13], URLs can be modified by content publishers, third parties linking to other sites, as well as by users. Furthermore, URLs can be rewritten through a service in our architecture. While Na Kika also supports static proxy configuration in browsers, we prefer URL rewriting as it allows for more fine-grained load balancing between edge nodes and presents a uniform, location-independent interface for using our architecture.

3.1 Programming Model

The functionality of hosted services and applications is specified through two event handlers, which are written in JavaScript. Our architecture does not depend on the choice of scripting language and could support several languages. We chose JavaScript because it already is widely used by web developers. Additionally, we found its C-like syntax and prototype-based object model helpful in writing scripts quickly and with little code; though we had to add support for byte arrays to avoid unnecessarily copying data. The onRequest event handler accepts an HTTP request and returns either a request for continued processing or a response representing the corresponding content or error condition. The onResponse event handler accepts an HTTP response and always returns a response. A pair of onRequest and onResponse event handlers mimics the high-level organization of any HTTP proxy and represents the unit of composition in Na Kika: a scripting pipeline stage.

In providing two interposition points for HTTP processing, *Na Kika* differs from other systems, such as Active Cache [9], SDT [19], and TACC [12], which only interpose on HTTP responses. Interposition on requests is necessary for HTTP redirection and, more importantly, as a first-line defense for enforcing access controls. It also is more efficient if responses are created from scratch, as it avoids accessing a resource before edge-side processing. To facilitate the secure composition of untrusted services, *Na Kika* relies on fewer event handlers than Pai et al.'s proxy API [31]; though it does provide similar expressivity, notably, to control the proxy cache, through its platform libraries.

Similar to ASP.NET and JSP, requests and responses are not passed as explicit arguments and return values, but are represented as global JavaScript objects. Using global objects provides a uniform model for accessing functionality and data, since native-code libraries, which we call vocabularies, also expose their functionality through global JavaScript objects. Na Kika provides vocabularies for managing HTTP messages and state and for performing common content processing steps. In particular, it provides support for accessing URL components, cookies, and the proxy cache, fetching other web resources, managing hard state, processing regular expressions, parsing and transforming XML documents, and transcoding images. We expect to add vocabularies for performing cryptographic operations and transcoding movies as well. Figure 2 illustrates an example onResponse event handler.

For HTTP responses, the body always represents the entire instance [25] of the HTTP resource, so that the resource can be correctly transcoded [19]. If the response represents an unmodified or partial resource, it is instantiated, for example, by retrieving it from the cache, when

```
onResponse = function() {
  var buff = null, body = new ByteArray();
  while (buff = Response.read()) {
   body.append(buff);
  var type = ImageTransformer.
    type (Response.contentType);
  var dim = ImageTransformer.
    dimensions (body, type);
  if (\dim x > 176 \mid | \dim y > 208) {
    var ima;
    if (\dim x/176 > \dim y/208) {
      img = ImageTransformer.transform(body,
        type, "jpeg", 176, dim.y/dim.x*208);
      img = ImageTransformer.transform(body.
        type, "jpeg", dim.x/dim.y*176, 208);
    Response.setHeader("Content-Type",
      "image/jpeg");
    Response.setHeader("Content-Length",
      ima.length);
    Response.write(img);
```

Figure 2: An example onResponse event handler, which transcodes images to fit onto the 176 by 208 pixel screen of a Nokia cell phone. It relies on the image transformer vocabulary to do the actual transcoding. The response body is accessed in chunks to enable cut-through routing; though the transformer vocabulary does not yet support it, with the script buffering the entire body.

a script accesses the body.

Event Handler Selection

To provide script modularity and make individual pipeline stages easily modifiable, stages do not consist of a fixed pair of event handlers; rather, the particular event handlers to be executed for each stage are selected from a collection of event handlers. To facilitate this selection process, pairs of onRequest and onResponse event handlers are associated with predicates on HTTP requests, including, for example, the client's IP address or the resource's URL. Conceptually, *Na Kika* first evaluates all of a stage's predicates and then selects the pair with the closest valid match for execution.

The association between event handlers and predicates is expressed in JavaScript by instantiating *policy objects*. As illustrated in Figure 3, each policy object has several properties that contain a list of allowable values for the corresponding HTTP message fields. Each policy object also has two properties for the onRequest and onResponse event handlers and an optional nextStages property for scheduling additional stages as discussed below. Lists of allowable values support prefixes for URLs, CIDR notation for IP

Figure 3: An example policy object. The policy applies the onResponse event handler to all content on servers at NYU's or University of Pittsburgh's medical schools accessed from within the two universities. The call to register() activates the policy.

addresses, and regular expressions for arbitrary HTTP headers. When determining the closest valid match, different values in a property's list are treated as a disjunction, different properties in a policy object are treated as a conjunction, and null properties are treated as truth values. Furthermore, precedence is given to resource URLs, followed by client addresses, then HTTP methods, and finally arbitrary headers. Null event handlers are treated as no-ops for event handler execution, thus making it possible to process only requests or responses or to use a stage solely for scheduling other stages.

Selecting event handlers by declaring predicates on HTTP messages avoids long sequences of if-else statements in a single, top-level event handler, thus resulting in more modular event processing code. When compared to the additional HTTP headers used by Active Cache and SDT for selecting edge-side code, predicate-based script selection also enables the interposition of code not specified by the origin server, an essential requirement for both composing services and enforcing security. While designing Na Kika, we did consider a domainspecific language (DSL) for associating predicates with event handlers instead of using JavaScript-based policy objects. While a DSL can be more expressive (for example, by allowing disjunction between properties), we rejected this option because it adds too much complexity both for web developers targeting Na Kika and for implementors of our architecture-while providing little additional benefits. We also considered performing predicate selection on HTTP responses, but believe that pairing event handlers results in a simpler programming model, with little loss of expressivity. Also matching responses requires a very simple change to our implementation.

Scripting Pipeline Composition

By default, each scripting pipeline has three stages. The first stage provides administrative control over clients' access to our edge-side computing network. It can, for example, perform rate limiting, redirect requests, or reject them altogether. The second stage performs site-specific processing, which typically serves as a surrogate

```
procedure EXECUTE-PIPELINE(request)
   forward \leftarrow EMPTY
   backward \leftarrow EMPTY
   > Start with administrative control and site-specific stages
   PUSH(forward, "http://nakika.net/serverwall.js")
   PUSH(forward, SITE(request.url) + "/nakika.js")
   PUSH(forward, "http://nakika.net/clientwall.js")
                       > Schedule stages and execute onRequest
   repeat
       script \leftarrow Fetch-And-Execute(Pop(forward))
       policy \leftarrow FIND-CLOSEST-MATCH(script, request)
       Push(backward, policy)
       if policy.onRequest \neq NIL then
          response \leftarrow Run(policy.onRequest, request)
          > If handler creates response, reverse direction
          if response \neq NIL then exit repeat end if
       if policy.nextStages \neq NIL then
                                                ▶ Add new stages
          PREPEND(forward, policy.nextStages)
       end if
   until forward = EMPTY
   if response = NIL then
                                         > Fetch original resource
       response \leftarrow Fetch(request)
                                         repeat
       policy \leftarrow Pop(backward)
       if policy.onResponse \neq NIL then
          Run(policy.onResponse, response)
       end if
   until backward = EMPTY
   return response
end procedure
```

Figure 4: Algorithm for executing a pipeline. The algorithm interleaves computing a pipeline's schedule with onRequest event handler execution, so that matching can take into an account when an event handler modifies the request, notably to redirect it.

for the origin server and actually creates dynamic content. For example, this stage adapts medical content in a web-based educational environment to a students' learning needs. The third stage provides administrative control over hosted scripts' access to web resources. Similar to the first stage, it can redirect or reject requests.

To perform additional processing, each pipeline stage can dynamically schedule further stages by listing the corresponding scripts in a policy object's nextStages property. As shown in Figure 4, the dynamically scheduled stages are placed directly after the scheduling stage but before other, already scheduled stages. A sitespecific script can thus delay content creation until a later, dynamically scheduled stage, while also scheduling additional processing before that stage. Examples for such intermediate services include providing annotations (electronic post-it notes) for textual content and transcoding movies for access from mobile devices. To put it differently, each site can configure its own pipeline and thus has full control over how its content is created and transformed-within the bounds of Na Kika's administrative control. At the same time, new services,

such as visualization of the spread of diseases, can easily be layered on top of existing services, such as geographical mapping, even when the services are provided by different sites: the new service simply adjusts the request, including the URL, and then schedules the original service after itself. Both services are executed within a single pipeline on the same *Na Kika* node.

The scripts for each stage are named through regular URLs, accessed through regular HTTP, and subject to regular HTTP caching. As shown in Figure 4, the administrative control scripts are accessed from well-known locations; though administrators of *Na Kika* nodes may override these defaults to enforce their own, location-specific security controls. Site-specific scripts are accessed relative to the server's domain, in a file named nakika.js, which is comparable to the use of robots.txt and favicon.ico for controlling web spiders and browser icons, respectively. All other services, that is, dynamically scheduled pipeline stages, can be hosted at any web location and are accessed through their respective URLs.

In combining content creation with content transformation, our architecture's scripting pipeline is reminiscent of the Apache web server and Java servlets. At the same time, both Apache and Java servlets have a more complicated structure. They first process a request through a chain of input filters, then create a response in a dedicated module (the content handler for Apache and the actual servlet for Java servlets), and finally process the response through a chain of output filters. In mirroring an HTTP proxy's high-level organization, Na Kika's scripting pipeline stages have a simpler interface-requiring only two event handlers-and are also more flexible, as any onRequest event handler can generate a response. Furthermore, the content processing pipelines for Apache and Java servlets can only be configured by code outside the pipelines, while each stage in Na Kika's scripting pipelines can locally schedule additional stages—with the overall result that Na Kika is more flexible and more easily extensible, even in the presence of untrusted code.

Na Kika Pages

While our architecture's event-based programming model is simple and flexible, a large portion of dynamic content on the web is created by markup-based content management systems, such as PHP, JSP, and ASP.NET. To support web developers versed in these technologies, *Na Kika* includes an alternative programming model for site-specific content. Under this model, HTTP resources with the nkp extension or text/nkp MIME type are subject to edge-side processing: all text between the <?nkp start and ?> end tags is treated as JavaScript and replaced by the output of running that code. These

```
bmj = "bmj.bmjjournals.com/cgi/reprint";
nejm = "content.nejm.org/cgi/reprint";
p = new Policy();
p.url = [ bmj, nejm ];

p.onRequest = function() {
  if (! System.isLocal(Request.clientIP)) {
    Request.terminate(401);
  }
}
p.register();
```

Figure 5: An example policy that prevents access to the digital libraries of the BMJ (British Medical Journal) and the New England Journal of Medicine from clients outside a *Na Kika* node's hosting organization. The 401 HTTP error code indicates an unauthorized access.

so-called *Na Kika Pages* are implemented on top of *Na Kika*'s event-based programming model through a simple, 60 line script. We expect to utilize a similar technique to also support edge side includes [29, 44] (ESI) within the *Na Kika* architecture.

3.2 Security and Resource Controls

Na Kika's security and resource controls need to protect (1) the proxies in our edge-side computing network against client-initiated exploits, such as those encountered by CoDeeN [48], (2) the proxies against exploits launched by hosted code, and (3) other web servers against exploits carried through our architecture. We address these three classes of threats through admission control by the client-side administrative control stage, resource controls for hosted code, and emission control by the server-side administrative control stage, respectively. Of course, it is desirable to drop requests early, before resources have been expended [26], and, consequently, requests that are known to cause violations of Na Kika's security and resource controls should always be rejected at the client-side administrative control stage.

Because the two administrative control stages mediate all HTTP requests and responses entering and leaving the system, they can perform access control based on client and server names as well as rate limiting based on request rates and response sizes. The corresponding policies are specified as regular scripts and can thus leverage the full expressivity of *Na Kika*'s predicate matching. For instance, Figure 5 shows a policy object rejecting unauthorized accesses to digital libraries, which is one type of exploit encountered by CoDeeN. For more flexibility, security policies can also leverage dynamically scheduled stages. For example, the two administrative control stages can delegate content blocking to separate stages whose code, in turn, is dynamically created by a script based on a blacklist.

To enforce resource controls, a resource manager

```
procedure CONTROL(resource)
   priorityq \leftarrow EMPTY
   if Is-Congested(resource) then
                                     > Track usage and throttle
      for site in ACTIVE-SITES() do
          UPDATE(site.usage, resource)
          ENQUEUE(priorityq, site)
          THROTTLE(site, resource)
   else if ¬ Is-RENEWABLE(resource) then
                                                ▶ Track usage
      for site in ACTIVE-SITES() do
          UPDATE(site.usage, resource)
      end for
   end if
   WAIT(TIMEOUT)

    ▶ Let throttling take effect

   if Is-Congested(resource) then
      TERMINATE(DEQUEUE(priorityq))
                                            UNTHROTTLE(resource)
                                     > Restore normal operation
   end if
end procedure
```

Figure 6: Algorithm for congestion control. The CONTROL procedure is periodically executed for each tracked resource. Note that our implementation does not block but rather polls to detect timeouts.

tracks CPU, memory, and bandwidth consumption as well as running time and total bytes transferred for each site's pipelines. It also tracks overall consumption for the entire node. As shown in Figure 6, if any of these resources is overutilized, the resource manager starts throttling requests proportionally to a site's contribution to congestion and, if congestion persists, terminates the pipelines of the largest contributors. A site's contribution to congestion captures the portion of resources consumed by its pipelines. For renewable resources, i.e., CPU, memory, and bandwidth, only consumption under overutilization is included. For nonrenewable resources, i.e., running time and total bytes transferred, all consumption is included. In either case, the actual value is the weighted average of past and present consumption and is exposed to scripts—thus allowing scripts to adapt to system congestion and recover from past penalization.

To complete resource controls, all pipelines are fully sandboxed. They are isolated from each other, running, for example, with their own heaps, and can only access select platform functionality. In particular, all regular operating system services, such as processes, files, or sockets, are inaccessible. The only resources besides computing power and memory accessible by scripts are the services provided by *Na Kika*'s vocabularies (that is, native-code libraries).

We believe that *Na Kika*'s congestion-based resource management model is more appropriate for open systems than more conventional quota-based resource controls for two reasons. First, open systems such as *Na Kika* have a different usage model than more conven-

tional hosting platforms: they are open to all content producers and consumers, with hosting organizations effectively donating their resources to the public. In other words, services and applications should be able to consume as many resources as they require—as long as they do not interfere with other services, i.e., cause congestion. Second, quota-based resource controls require an administrative decision as to what resource utilization is legitimate. However, even when quotas are set relative to content size [9], it is hard to determine appropriate constants, as the resource requirements may vary widely. We did consider setting fine-grained quotas through predicates on HTTP messages, comparable to how our architecture selects event handlers. However, while predicatebased policy selection is flexible, it also amplifies the administrative problem of which constants to choose for which code.

Our architecture's utilization of scripting has two advantages for security and resource control when compared to other edge-side systems. First, administrative control scripts simplify the development and deployment of security policy updates. Once a fix to a newly discovered exploit or abuse has been implemented, the updated scripts are simply published on the Na Kika web site and automatically installed across all nodes when cached copies of the old scripts expire. In contrast, CoDeeN and other edge-side systems that hard code security policies require redistribution of the system binaries across all nodes. Though Na Kika still requires binary redistribution to fix security holes in native code. Second, providing assurance that hosted services and applications are effectively secured is simpler for scripts than for Java or native code. Our starting point is a bare scripting engine to which we selectively add functionality, through vocabularies, rather than trying to restrict a general purpose platform after the fact.

3.3 Hard State

The web's expiration-based consistency model for cached state is sufficient to support a range of edge-side applications, including content assembly (through, for example, edge-side includes [29, 44]) or the transcoding of multi-media content. However, a complete platform for edge-side content management also requires support for managing hard state such as edge-side access logs and replicated application state. Edge-side logging provides accurate usage statistics to content producers, while edge-side replication avoids accessing the origin server for every data item.

Na Kika performs access logging on a per-site basis. Logging is triggered through a site's script, which specifies the URL for posting log updates. Periodically, each Na Kika node scans its log, collects all entries for each specific site, and posts those portions of the log to the

specified URLs.

Na Kika's support for edge-side replication builds on Gao et al.'s use of distributed objects, which, internally, rely on domain-specific replication strategies to synchronize state updates and support both pessimistic and optimistic replication [14]. Like Gao et al., Na Kika's hard state replication relies on a database for local storage and a reliable messaging service for propagating updates, which are exposed through vocabularies. Unlike Gao et al., Na Kika's hard state replication is implemented by regular scripts. Updates are accepted by a script, written to local storage, and then propagated to other nodes through the messaging layer. Upon receipt of a message on another node, a regular script processes the message and applies the update to that node's local storage. As a result, Na Kika provides content producers with considerable flexibility in implementing their domain-specific replication strategies. For example, the script accepting updates can propagate them only to the origin server to ensure serializability or to all nodes to maximize availability. Furthermore, the script accepting messages can easily implement domain-specific conflict resolution strategies. To secure replicated state, Na Kika partitions hard state amongst sites and enforces resource constraints on persistent storage. Since update processing is performed by regular scripts, it already is subject to Na Kika's security and resource controls.

3.4 Overlay Network

The Na Kika architecture relies on a structured overlay network for coordinating local caches and for enabling incremental deployment with low administrative overhead. From an architectural viewpoint, the overlay is treated largely as a black box, to be provided by an existing DHT [13, 16, 35, 42, 52]. This reflects a conscious design decision on our end and provides us with a test case for whether DHTs can, in fact, serve as robust and scalable building blocks for a global-scale distributed system. Our prototype implementation builds on Coral [13], which is well-suited to the needs of our architecture, as Coral explicitly targets soft state and includes optional support for DNS redirection to local nodes. As we deploy Na Kika, we expect to revisit the functionality provided by the DHT. Notably, load balancing, which is currently provided at the DNS level, can likely benefit from application-specific knowledge, such as the number of concurrent HTTP exchanges being processed by a node's scripting pipelines.

3.5 Summary

The Na Kika architecture leverages scripting and overlay networks to provide an open edge-side computing network. First, Na Kika exposes a programming model already familiar to web developers by organizing hosted services and applications into a pipeline of scripted event handlers that process HTTP requests and responses. Second, it provides a secure execution platform by mediating all HTTP processing under administrative control, by isolating scripts from each other, and by limiting resource utilization based on overall system congestion. Third, it provides extensibility by dynamically scheduling event handlers within a pipeline stage as well as additional pipeline stages through predicate matching. Finally, it provides scalability by organizing all nodes into an automatically configured overlay network, which supports the redirection of clients to (nearby) edge nodes and the addition of new nodes with low administrative overhead.

At the same time, web integration is not entirely complete, as URLs need to be rewritten for Na Kika access. As already discussed, URLs can be automatically rewritten by web browsers, hosted code, as well as servers and, consequently, the need for manual rewriting will decrease over time. Furthermore, while our architecture protects against client- and script-initiated exploits, it does not currently protect against misbehaving edgeside nodes. In particular, nodes can arbitrarily modify cached content, which is especially problematic for administrative control scripts. We return to the issue of content integrity in Section 6.

4 Implementation

Our prototype implementation of Na Kika builds on three open source packages: the Apache 2.0 web server, the Mozilla project's SpiderMonkey JavaScript engine [27], and the Coral distributed hashtable [13]. We chose Apache for HTTP processing because it represents a mature and cross-platform web server. Similarly, Spider-Monkey is a mature and cross-platform implementation of JavaScript and used across the Mozilla project's web browsers. Additionally, our prototype includes a preliminary implementation of hard state replication, which relies on the Java-based JORAM messaging service [30] and exposes a vocabulary for managing user registrations, as required by the SPECweb99 benchmark. Our implementation adds approximately 23,000 lines of C code to the 263,000 lines of code in Apache, the 123,000 lines in SpiderMonkey, and the 60,000 lines in Coral. The majority of changes is to Apache and mostly contained in Apache modules. Our modified Apache binary, including dynamically loaded libraries, is 10.6 MByte large and the Coral DHT server is 13 MByte.

As already mentioned in Section 3.1, Apache structures HTTP processing into a chain of input filters that operate on requests, followed by a content handler that generates responses, followed by a chain of output filters that operate on responses. Our prototype implements the scripting pipeline by breaking each stage

into a pair of input and output filters, which execute the onRequest and onResponse event handlers, respectively, and by dynamically inserting the pair into Apache's filter chain. The content handler is a modified version of Apache's mod_proxy, which implements the proxy cache and, in our version, also interfaces with the DHT. If an onRequest event handler generates an HTTP response, our implementation sets a flag that prevents the execution of scripts in later pipeline stages and of the proxy caching code, while still conforming with Apache's sequencing of input filters, content handler, and output filters.

To provide isolation, our implementation executes each pipeline in its own process and each script, in turn, in its own user-level thread and with its own scripting context, including heap. Scripting contexts are reused to amortize the overhead of context creation across several event handler executions; this is safe because JavaScript programs cannot forge pointers and the heap is automatically garbage collected. A separate monitoring process tracks each pipeline's CPU, memory, and network consumption and periodically executes the congestion control algorithm in Figure 6. To throttle a site's pipelines, the monitoring process sets a flag in shared memory, which causes the regular Apache processes to reject requests for that site's content with a server busy error. To terminate a site's pipelines, the monitoring process kills the corresponding Apache processes, thus putting an immediate stop to processing even if a pipeline is executing a vocabulary's native code.

Employing per-script user-level threads also helps integrate script execution with Apache, while still exposing a simple programming model. In particular, Apache's sequence of input filters, content handler, and output filters is not necessarily invoked on complete HTTP requests and responses. Rather, each filter is invoked on chunks of data, the so-called bucket brigades, as that data becomes available. As a result, Apache may interleave the execution of several onRequest and onResponse event handlers. Per-script user-level threads hide this piecemeal HTTP processing from script developers, providing the illusion of scripts running to completion before invoking the next stage. To avoid copying data between Apache and the scripting engine, our implementation adds byte arrays as a new core data type to SpiderMonkey. Whenever possible, these byte arrays directly reference the corresponding bucket brigade buffers.

The policy matching code trades off space for dynamic predicate evaluation performance. While loading a script and registering policy objects, the matcher builds a decision tree for that pipeline stage, with nodes in the tree representing choices. Starting from the root of the tree, the nodes represent the components of a resource URL's server name, the port, the components of the path, the

Name	Description
Proxy	A regular Apache proxy.
DHT	The proxy with an integrated DHT.
Admin	A Na Kika node evaluating one matching predicate and executing empty event handlers for each of the two administrative control stages.
Pred-n	The Admin configuration plus another stage evaluating predicates for <i>n</i> policy objects, with no matches.
Match-1	The Admin configuration plus another stage evaluating one matching predicate and executing the corresponding, empty event handlers.

Table 1: The different micro-benchmark configurations.

components of the client address, the HTTP methods, and, finally, individual headers. If a property of a policy object does not contain any values, the corresponding nodes are skipped. Furthermore, if a property contains multiple values, nodes are added along multiple paths. When all properties have been added to the decision tree, the event handlers are added to the current nodes, once for each path. With the decision tree in place, dynamic predicate evaluation simply is a depth-first search across the tree for the node closest to the leaves that also references an appropriate event handler. Decision trees are cached in a dedicated in-memory cache. The implementation also caches the fact that a site does *not* publish a policy script, thus avoiding repeated checks for the nakika.js resource.

5 Evaluation

To evaluate Na Kika, we performed a set of local microbenchmarks and a set of end-to-end experiments, which include wide area experiments on the PlanetLab distributed testbed [32]. The micro-benchmarks characterize (1) the overhead introduced by Na Kika's DHT and scripting pipeline and (2) the effectiveness of our congestion-based resource controls. The end-to-end experiments characterize the performance and scalability of a real-world application and of a modified SPECweb99 benchmark. We also implemented three new services to characterize the extensibility of our edge-side computing network. In summary, our experimental results show that, even though the scripting pipeline introduces noticeable overheads, Na Kika is an effective substrate both for scaling web-based applications and for extending them with new functionality.

5.1 Micro-Benchmarks

To characterize the overheads introduced by *Na Kika*'s DHT and scripting pipeline, we compare the performance of a *Na Kika* node with a regular Apache proxy cache for accessing a single, static 2,096 byte document representing Google's home page (without in-

Configuration	Cold Cache	Warm Cache	
Proxy	3	1	
DHT	5	1	
Admin	16	2	
Pred-0	19	2	
Pred-1	20	2	
Match-1	21	2	
Pred-10	22	2	
Pred-50	30	2	
Pred-100	41	2	

Table 2: Latency in milliseconds for accessing a static page under the different configurations.

line images). Since static resources are already well-served by existing proxy caches and CDNs, these microbenchmarks represent a worst-case usage scenario for *Na Kika*. After all, any time spent in the DHT or in the scripting pipeline adds unnecessary overhead. For all experiments, we measured the total time of a client accessing the static web page through a proxy—with client, proxy, and server being located on the same, switched 100 Mbit ethernet. The proxy runs on a Linux PC with a 2.8 GHz Intel Pentium 4 and 1 GB of RAM.

We performed 18 experiments, representing 9 different configurations under both a cold and a warm proxy cache. The different configurations are summarized in Table 1 and determine the overhead of DHT integration, baseline administrative control, predicate matching, and event handler invocation, respectively. For the cold cache case of the *Admin*, *Pred-n*, and *Match-1* configurations, the administrative control and site-specific scripts are fetched from the local server and evaluated to produce the corresponding decision tree. For the warm cache case, the cached decision tree is used. Resource control is disabled for these experiments.

Table 2 shows the latency in milliseconds for the 18 different experiments. Each number is the average of 10 individual measurements. Overall, the results clearly illustrate the basic cost of utilizing Na Kika: its scripting pipeline. For the Pred-n and Match-1 configurations under a cold cache, loading the actual page takes 2.9 ms and loading the script takes between 2.5 ms and 5.6 ms, depending on size. Additionally, the creation of a scripting context takes 1.5 ms. Finally, parsing and executing the script file takes between 0.08 ms and 17.8 ms, again, depending on size. However, the results also illustrate that our implementation's use of cachingfor resources, scripting contexts, and decision trees-is effective. Retrieving a resource from Apache's cache takes 1.1 ms and retrieving a decision tree from the inmemory cache takes 4 µs. Re-using a scripting context takes 3 μ s. Finally, predicate evaluation takes less than 38 μ s for all configurations. However, these operations also result in a higher CPU load: the Na Kika

node reaches capacity with 30 load-generating clients at 294 requests/second (rps) under *Match-1*, while the plain Apache proxy reaches capacity with 90 clients at 603 rps on the same hardware. Since both resources and scripts only need to be accessed when reaching their expiration times, we expect that real world performance is closer to warm cache than cold cache results. Furthermore, most web resources are considerably larger than Google's home page, so that network transfer times will dominate scripting pipeline latency.

Resource Controls

To characterize the effectiveness of Na Kika's congestion-based resource management, we compare the performance of a Na Kika node with and without resource controls under high load, such as that caused by a flash crowd. For these experiments, the Na Kika proxy runs on the same Linux PC as before. Load is generated by accessing the same 2,096 byte page under the Match-1 configuration in a tight loop. With 30 load generators (i.e., at the proxy's capacity), we measure 294 rps without and 396 rps with resource controls. With 90 load generators (i.e., under overload), we measure 229 rps without and 356 rps with resource controls. If we also add one instance of a misbehaving script, which consumes all available memory by repeatedly doubling a string, the throughput with 30 load generators drops to 47 rps without but only 382 rps with resource controls. For all experiments, the runs with resource controls reject less than 0.55% of all offered requests due to throttling and drop less than 0.08% due to termination, including the one triggering the misbehaving script. These results illustrate the benefits of Na Kika's resource controls. Even though resource management is reactive, throttling is effective at ensuring that admitted requests have sufficient resources to run to completion, and termination is effective at isolating the regular load from the misbehaving one.

5.2 Web-based Medical Education

To evaluate a real-world application running on *Na Kika*, we compare the Surgical Interactive Multimedia Modules [43] (SIMMs) in their original single-server configuration with an initial port to our edge-side computing network. The SIMMs are a web-based educational environment that is being developed by NYU's medical school. Each SIMM focuses on a particular medical condition and covers the complete workup of a patient from presentation to treatment to follow-up. It consists of rich-media enhanced lectures, annotated imaging studies, pathology data, and animated and real-life surgical footage—comprising around 1 GB of multimedia content per module. The five existing SIMMs already are an integral part of the curriculum at NYU's medical school and are also used at four other medical schools in

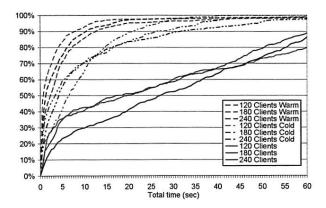


Figure 7: Cumulative distribution function (CDF) for latency to access HTML content in the SIMMs' single-server and *Na Kika* cold and warm cache configurations.

the U.S. and Australia, with more institutions to follow. The SIMMs rely heavily on personalized and multimedia content but do not contain any real patient data (with its correspondent privacy requirements), thus making them good candidates for deployment on *Na Kika*.

The SIMMs are implemented on top of Apache Tomcat 5.0 and MySQL 4.1. They utilize JSP and Java servlets to customize content for each student as well as to track her progress through the material and the results of sectional assessments. To facilitate future interface changes as well as different user interfaces, customized content is represented as XML and, before being returned to the client, rendered as HTML by an XSL stylesheet (which is the same for all students)². The initial *Na Kika* port off-loads the distribution of multimedia content, since it is large, and the (generic) rendering of XML to HTML, since it is processor intensive, to our edge-side computing network. Content personalization is still performed by the central server; we expect that future versions will also move personalization to the edge.

The port was performed by one of the main developers of the SIMMs and took two days. The developer spent four hours on the actual port—which entailed changing URLs to utilize *Na Kika*, making XML and XSL content accessible over the web, and switching from cookies to URL-based session identifiers as well as from HTTP POSTs to GETs—and the rest of the two days debugging the port. In fact, the main impediment to a faster port was the relative lack of debugging tools for our prototype implementation. The port adds 65 lines of code to the existing code base of 1,900 lines, changes 25 lines, and removes 40 lines. The new nakika.js policy consists of 100 lines of JavaScript code.

To evaluate end-to-end performance, we compare the

single-server version with the Na Kika port accessed through a single, local proxy-which lets us compare baseline performance—and with the Na Kika port running on proxies distributed across the wide area—which lets us compare scalability. For all experiments, we measure the total time to access HTML content-which represents client-perceived latency-and the average bandwidth when accessing multimedia files-which determines whether playback is uninterrupted. Load is generated by replaying access logs for the SIMMs collected by NYU's medical school; log replay is accelerated 4× to produce noticeable activity. For the local experiments, we rely on four load-generating nodes. For the widearea experiments, 12 load-generating PlanetLab nodes are distributed across the U.S. East Coast, West Coast, and Asia—thus simulating a geographically diverse student population—and, for Na Kika, matched with nearby proxy nodes. For Na Kika, we direct clients to randomly chosen, but close-by proxies from a preconfigured list of node locations. For the local experiments, the origin server is the same PC as used in Section 5.1; for the widearea experiments, it is a PlanetLab node in New York.

The local experiments show that, under a cold cache and heavy load, the performance of the single Na Kika proxy trails that of the single server. Notably, for 160 clients (i.e., 40 instances of the log replay program running on each of 4 machines), the 90th percentile latency for accessing HTML content is 904 ms for the single server and 964 ms for the Na Kika proxy. The fraction of accesses to multimedia content consistently seeing a bandwidth of at least 140 Kbps—the SIMMs' video bitrate-is 100% for both configurations. However, when adding an artificial network delay of 80 ms and bandwidth cap of 8 Mbps between the server on one side and the proxy and clients on the other side (to simulate a wide-area network), the single Na Kika proxy already outperforms the single server, illustrating the advantages of placing proxies close to clients. For 160 clients, the 90th percentile latency for HTML content is 8.88 s for the single server and 1.21 s for the Na Kika proxy. Furthermore, only 26.2% of clients see sufficient bandwidth for accessing video content for the single server, while 99.9% do for the Na Kika proxy. As illustrated in Figure 7, the advantages of our edgeside computing network become more pronounced for the wide-area experiments. For 240 clients (i.e., 20 programs running on each of 12 machines), the 90th percentile latency for accessing HTML content is 60.1 s for the single server, 31.6 s for Na Kika with a cold cache, and 9.7 s with a warm cache. For the single server, the fraction of clients seeing sufficient video bandwidth is 0% and the video failure rate is 60.0%. For Na Kika with a cold cache, the fraction is 11.5% and the failure rate is 5.6%. With a warm cache, the fraction is 80.3% and

²An earlier version relied on a custom-built Macromedia Director client for rendering XML. It was abandoned in favor of regular web browsers due to the extra effort of maintaining a dedicated client.

the failure rate is 1.9%. For *Na Kika*, accesses to multimedia content benefit to a greater extent from a warm cache than accesses to HTML, since PlanetLab limits the bandwidth available to each hosted project.

5.3 Hard State Replication

To further evaluate end-to-end performance in the wide area, we compare the performance of a single Apache PHP server and the same server supported by Na Kika running a modified version of the SPECweb99 benchmark. For this experiment, we re-implemented SPECweb99's server-side scripts in PHP and Na Kika Pages. The single-server version relies on PHP because it is the most popular add-on for creating dynamic content to the most popular web server [36, 37]. The Na Kika version relies on replicated hard state to manage SPECweb99's user registrations and profiles. With clients and five Na Kika nodes on the U.S. West Coast and the server located on the East Coast, 80% dynamic requests, 160 simultaneous connections, and a runtime of 20 minutes, the PHP server has a mean response time of 13.7 s and a throughput of 10.8 rps. With a cold cache, the Na Kika version has a response time of 4.3 s and a throughput of 34.3 rps. Additional experiments show that the results are very sensitive to PlanetLab CPU load, thus indicating that Na Kika's main benefit for these experiments is the additional CPU capacity under heavy load (and, conversely, that Na Kika requires ample CPU resources to be effective). Our SPECweb99 compliance score is 0 due to the limited bandwidth available between PlanetLab nodes. Nonetheless, this benchmark shows that Na Kika can effectively scale a complex workload that includes static content, dynamic content, and distributed updates.

5.4 Extensions

To evaluate *Na Kika*'s extensibility, we implemented three extensions in addition to the *Na Kika Pages* extension discussed in Section 3.1: electronic annotations for the SIMMs, image transcoding for small devices, and content blocking based on blacklists. As described below, our experiences with these extensions confirm that *Na Kika* is, in fact, easily extensible. In particular, they illustrate the utility of predicate-based event handler selection and dynamically scheduled pipeline stages. Furthermore, they illustrate that developers can build useful extensions quickly, even if they are not familiar with *Na Kika* or JavaScript programming.

Our first extension adds electronic annotations, i.e., post-it notes, to the SIMMs, thus providing another layer of personalization to this web-based educational environment. The extended SIMMs are hosted by a site outside NYU's medical school and utilize dynamically scheduled pipeline stages to incorporate the *Na Kika* version

of the SIMMs. The new functionality supports electronic annotations by injecting the corresponding dynamic HTML into the SIMMs' HTML content. It also rewrites request URLs to refer to the original content and URLs embedded in HTML to refer to itself, thus interposing itself onto the SIMMs. The resulting pipeline has three non-administrative stages, one each for URL rewriting, annotations, and the SIMMs. The annotations themselves are stored on the site hosting the extended version. This extension took one developer 5 hours to write and debug and comprises 50 lines of code; it leverages a previously developed implementation of electronic annotations, which comprises 180 lines of code.

In contrast to the extension for electronic annotations, which represents one site building on another site's service, our second extension represents a service to be published on the web for use by the larger community. This extension scales images to fit on the screen of a Nokia cell phone and generalizes the onResponse event handler shown in Figure 2 to cache transformed content. The extension can easily be modified to support other types and brands of small devices by (1) parameterizing the event handler's screen size and (2) adding new policy objects that match other devices' User-Agent HTTP headers. This extension took a novice JavaScript developer less than two hours to write and debug and comprises 80 lines of code.

Our third extension does not provide new functionality, but rather extends Na Kika's security policy with the ability to block sites based on blacklists. Its intended use is to deny access to illegal content through Na Kika. The extension is implemented through two dynamically scheduled pipeline stages. The first new stage relies on a static script to dynamically generate the JavaScript code for the second new stage, which, in turn, blocks access to the URLs appearing on the blacklist. The static script reads the blacklist from a preconfigured URL and then generates a policy object for each URL appearing on that blacklist. The onRequest event handler for all policy objects is the same handler, denying access as illustrated in Figure 5. This extension took 4.5 hours to write and debug, with an additional 1.5 hours for setting up a testbed. Since this extension represents the developer's first Na Kika as well as JavaScript code, the 4.5 hours include one hour mostly spent familiarizing himself with JavaScript. The extension comprises 70 lines of code.

6 Discussion and Future Work

As presented in this paper, *Na Kika* assumes that edgeside nodes are trusted, which effectively limits the organizations participating in a deployment. To allow *Na Kika* to scale to a larger number of edge networks and nodes, we are currently working towards eliminating this requirement by automatically ensuring the integrity

of content served by our edge-side computing network. Content integrity is important for producers and consumers so that, for example, the results of medical studies cannot be falsified. It also is important for the operation of the network itself, as scripts, including those used for administrative control, are accessed through and cached within *Na Kika*.

For original content, protecting against inadvertent or malicious modification reduces to detecting such changes and then retrieving the authoritative version from the origin server. However, using cryptographic hashes, for example, through self-certifying pathnames [24] as suggested in [13], is insufficient, as they cannot ensure freshness. To provide both integrity and freshness, we have already implemented an alternative solution that integrates with HTTP's cache control by adding two new headers to HTTP responses. The X-Content-SHA256 header specifies a cryptographic hash of the content for integrity and, to reduce load, can be precomputed. The X-Signature header specifies a signature over the content hash and the cache control headers for freshness. Our solution requires the use of absolute cache expiration times instead of the relative times introduced in HTTP/1.1 [21] as nodes cannot be trusted to correctly decrement relative times.

For processed or generated content, content integrity cannot be established through hashes and signatures alone, as content processing is performed by potentially untrusted nodes. Instead, we are exploring a probabilistic verification model. Under this model, a trusted registry maintains *Na Kika* membership. To detect misbehaving nodes, clients forward a fraction of content received from *Na Kika* proxies to other proxies, which then repeat any processing themselves. If the two versions do not match, the original proxy is reported to the registry, which uses this information to evict misbehaving nodes from the edge-side computing network.

7 Conclusions

Edge-side content management reduces load on origin servers, bandwidth consumption across the internet, and latency for clients. It also absorbs load spikes for underprovisioned servers. To make these benefits available to *all* content producers and consumers and thus to foster collaboration and innovation on web-based applications, *Na Kika* provides an open architecture for edge-side content creation, transformation, and caching.

Services and applications hosted by *Na Kika* are expressed through scripted event handlers. Event handlers are selected through predicates on HTTP messages and are composed into a pipeline that combines administrative control and site-specific processing. The resulting programming model is not only familiar to web developers versed in client-side scripting and the content pro-

cessing pipelines of Apache and Java servlets, but it also is more secure and more easily extensible. To provide security, Na Kika's scripting pipeline mediates all requests and responses passing through the system. Furthermore, all hosted services and applications are isolated from each other and the underlying operating system and subject to congestion-based resource management: hosted code can consume resources without restriction as long as it does not cause overutilization. To provide incremental scalability, all Na Kika nodes are organized into a structured overlay network, which enables DNS redirection of clients to nearby nodes and cooperative caching of both original and processed content. The experimental evaluation demonstrates that Na Kika's prototype implementation is effective at reducing load on origin servers and latency for clients, supporting significantly larger user populations than a single dynamic web server. It also demonstrates that Na Kika is, in fact, easily programmable and extensible.

Acknowledgments

Bill Holloway ported the SIMMs to *Na Kika* and Jake Aviles helped with their evaluation. Robert Soule implemented static content integrity and the security policy extension. Our shepherd, Emin Gün Sirer, and the anonymous reviewers provided valuable feedback on earlier versions of this paper. This material is based in part upon work supported by the National Science Foundation under Grant No. 0537252 and by the New York Software Industry Association.

References

- Akamai Technologies, Inc. A developer's guide to on-demand distributed computing, Mar. 2004.
- [2] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proc. 7th IWCW*, Aug. 2002.
- [3] A. Barbir, O. Batuner, B. Srinivas, M. Hofmann, and H. Orman. Security threats and risks for open pluggable edge services (OPES). RFC 3837, IETF, Aug. 2004.
- [4] A. Barbir, R. Penno, R. Chen, M. Hofmann, and H. Orman. An architecture for open pluggable edge services (OPES). RFC 3835, IETF, Aug. 2004.
- [5] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large web site population with implications for content delivery. In *Proc. 13th WWW*, pp. 522–533, May 2004.
- [6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. D. Jensen, eds., Secure Internet Programming, vol. 1603 of LNCS, pp. 185–210. Springer, 1999.
- [7] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proc. 17th NCSC*, pp. 18–27, 1985.
- [8] C. Canali, V. Cardellini, M. Colajanni, R. Lancellotti, and P. S. Yu. Cooperative architectures and algorithms for discovery and transcoding of multi-version content. In *Proc. 8th IWCW*, Sept. 2003.
- [9] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents on the web. In *Proc. Middleware '98*, pp. 373–388, Sept. 1998.

- [10] D. M. D'Alessandro, T. E. Lewis, and M. P. D'Alessandro. A pediatric digital storytelling system for third year medical students: The virtual pediatric patients. *BMC Medical Education*, 4(10), July 2004.
- [11] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, IETF, Sept. 1999.
- [12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th* SOSP, pp. 78–91, Oct. 1997.
- [13] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, Mar. 2004.
- [14] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. 12th WWW*, pp. 449–460, May 2003.
- [15] R. Grimm and B. N. Bershad. Separating access control policy, enforcement and functionality in extensible systems. ACM TOCS, 19(1):36–70, Feb. 2001.
- [16] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. 2nd IPTPS*, vol. 2735 of *LNCS*, pp. 160–169. Springer, Feb. 2003.
- [17] IBM Corporation. WebSphere Edge Server Administration Guide. 3rd edition, Dec. 2001.
- [18] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st PODC*, pp. 213–222, July 2002
- [19] B. Knutsson, H. Lu, J. Mogul, and B. Hopkins. Architecture and performance of server-directed transcoding. ACM TOIT, 3(4):392–424, Nov. 2003.
- [20] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. AGENT TCL: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, Jul./Aug. 1997.
- [21] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proc. 8th WWW*, May 1999.
- [22] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *Proc.* 2003 XMLSEC, pp. 25–37, 2003.
- [23] W.-Y. Ma, B. Shen, and J. Brassil. Content services network: The architecture and protocols. In *Proc. 6th IWCW*, June 2001.
- [24] D. Mazières and M. F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proc. 8th SIGOPS Europ. Workshop*, pp. 118–125, Sept. 1998.
- [25] J. C. Mogul. Clarifying the fundamentals of HTTP. In Proc. 11th WWW, pp. 25–36, May 2002.
- [26] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. ACM TOCS, 15(3):217–252, Aug. 1997.
- [27] Mozilla Foundation. SpiderMonkey (JavaScript-C) engine. http://www.mozilla.org/js/spidermonkey/.
- [28] National Center for Postsecondary Improvement. Beyond dead reckoning: Research priorities for redirecting American higher education. http://www.stanford.edu/group/ncpi/ documents/pdfs/beyond_dead_reckoning.pdf, Oct. 2002.
- [29] M. Nottingham and X. Liu. Edge architecture specification, 2001. http://www.esi.org/architecturespec_1-0.html.
- [30] ObjectWeb. JORAM. http://joram.objectweb.org/.
- [31] V. S. Pai, A. L. Cox, V. S. Pai, and W. Zwaenepoel. A flexible and efficient application programming interface for a customizable proxy cache. In *Proc. 4th USITS*, Mar. 2003.
- [32] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proc.* 1st HotNets, Oct. 2002.

- [33] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating internet applications. In *Proc.* 8th IWCW, Sept. 2003.
- [34] M. Rabinovich, Z. Xiao, F. Douglis, and C. Kalmanek. Moving edge-side includes to the real edge—the clients. In *Proc. 4th USITS*, Mar. 2003.
- [35] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, pp. 329–350, Nov. 2001.
- [36] Security Space. Apache module report. http: //www.securityspace.com/s_survey/data/man. 200501/apachemods.html, Feb. 2005.
- [37] Security Space. Web server survey. http://www.securityspace.com/s_survey/data/200501/index.html, Feb. 2005.
- [38] W. Shi, K. Shah, Y. Mao, and V. Chaudhary. Tuxedo: A peer-to-peer caching system. In *Proc.* 2003 PDPTA, pp. 981–987, June 2003.
- [39] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. 17th SOSP*, pp. 202–216, Dec. 1999.
- [40] E. G. Sirer and K. Wang. An access control language for web services. In *Proc. 7th SACMAT*, pp. 23–30, June 2002.
- [41] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proc. 4th USITS*, pp. 225–238, Mar. 2003.
- [42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. 2001 SIGCOMM*, pp. 149–160, Aug. 2001.
- [43] M. M. Triola, M. A. Hopkins, M. J. Weiner, W. Holloway, R. I. Levin, M. S. Nachbar, and T. S. Riles. Surgical interactive multimedia modules: A novel, non-browser based architecture for medical education. In *Proc. 17th CBMS*, pp. 423–427, June 2004.
- [44] M. Tsimelzon, B. Weihl, and L. Jacobs. ESI language specification 1.0, 2001. http://www.esi.org/language_spec_ 1-0.html.
- [45] S. H. J. Uijtdehaage, S. E. Dennis, and C. Candler. A web-based database for sharing educational multimedia within and among medical schools. *Academic Medicine*, 76:543–544, 2001.
- [46] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *Proc. 2nd USITS*, pp. 151–164, Oct. 1999.
- [47] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. 5th OSDI*, pp. 345–360, Dec. 2002.
- [48] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proc.* 2004 USENIX, pp. 171–184, June 2004.
- [49] S. A. Wartman. Research in medical education: The challenge for the next decade. Academic Medicine, 69(8):608–614, 1994.
- [50] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. 17th SOSP*, pp. 16–31, Dec. 1999.
- [51] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. 4th OSDI*, pp. 305–318, Oct. 2000.
- [52] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J-SAC*, 22(1):41–53, Jan. 2004.
- [53] W. Zhao and H. Schulzrinne. DotSlash: Providing dynamic scalability to web applications with on-demand distributed query result caching. Tech. Report CUCS-035-05, Columbia University, Sept. 2005.

Connection Conditioning: Architecture-Independent Support for Simple, Robust Servers

KyoungSoo Park and Vivek S. Pai Department of Computer Science Princeton University

Abstract

For many network server applications, extracting the maximum performance or scalability from the hardware may no longer be much of a concern, given today's pricing – a \$300 system can easily handle 100 Mbps of Web server traffic, which would cost nearly \$30,000 per month in most areas. Freed from worrying about absolute performance, we re-examine the design space for simplicity and security, and show that a design approach inspired by Unix pipes, Connection Conditioning (CC), can provide architecture-neutral support for these goals.

By moving security and connection management into separate filters outside the server program, CC supports multi-process, multi-threaded, and event-driven servers, with no changes to programming style. These filters are customizable and reusable, making it easy to add security to any Web-based service. We show that CC-aided servers can support a range of security policies, and that offloading connection management allows even simple servers to perform comparably to much more complicated systems.

1 Introduction

Web server performance has greatly improved due to a number of factors, including raw hardware performance, operating systems improvements (zero copy, timing wheels [29], hashed PCBs), and parallel scale-out via load balancers [9, 11] and content distribution networks [2, 14]. Coupled with the slower improvements in network price/performance, extracting the maximum performance from hardware may not be a high priority for most Web sites. Hardware costs can be dwarfed by bandwidth costs – a \$300 system can easily handle 100 Mbps of Web traffic, which would cost \$30,000 per month for wide-area bandwidth in the USA. For most sites, the performance and scalability of the server software itself may not be major issues – if the site can afford bandwidth, it can likely afford the required hardware.

These factors may partly explain why the Apache Web server's market share has increased to 69% [17] despite a decade of server architecture research [8, 12, 13, 18, 30, 32] that has often produced much faster servers – with all of the other advances, Apache's simple process pool performs well enough for most sites. The benefits of cost, flexibility, and community support compensate for any loss in maximum performance. Some Web sites

may want higher performance per machine, but even the event-driven Zeus Web server, often the best performer in benchmarks, garners less than 2% of the market [17]

Given these observations and future hardware trends, we believe designers are better served by improving server simplicity and security. Deployed servers are still simple to attack in many ways, and while some server security research [6, 21] has addressed these problems, it implicitly assumes the use of event-driven programming styles, making its adoption by existing systems much harder. Even when the research can be generalized, it often requires modifying the code of each application to be secured, which can be time-consuming and error-prone.

To address these problems, we revisit the lessons of Unix pipes to decompose server processing in a system called Connection Conditioning (CC). Requests are bundled with their sockets and passed through a series of general-purpose user-level filters that provide connection management and security benefits without invasive changes to the main server. These filters allow common security and connection management policies to be shared across servers, resulting in simpler design for server writers, and more tested and stable code for filter writers. This design is also architecture-neutral – it can be used in multi-process, threaded, and event-driven servers.

We demonstrate Connection Conditioning in two ways: by demonstrating its design and security benefits for new servers, and by providing security benefits to existing servers. We build an extremely simple CC-aware Web server that handles only one request at a time by moving all connection management to filters. This approach allows this simple design to efficiently serve thousands of simultaneous connections, without explicitly worrying about unpredictable/unbounded delays and blocking. This server is ideal for environments that require some robustness, such as sensors, and is so small and simple that it can be understood within a few minutes.

Despite its size, this server handles a broad range of workloads while resisting DoS attacks that affect other servers, both commercial and experimental. Its performance is sufficient for many sites – it generally outperforms Apache as well as some research Web servers. Using the filters developed for this server, we can improve the security of the Apache Web server as well as a research server, Flash, with a tolerable performance impact.

2 Background

All server software architectures ultimately address how to handle multiple connections that can block in several places, sometimes for arbitrarily long periods. Using some form of multiplexing (in the OS, the thread library, or at application level), these schemes try to keep the CPU utilized even when requests block or clients download data at different speeds. Blocking stems from two sources, network and disk, with disk being the more predictable source. Since the client is not under the server's control, any communication with it can cause network blocking. Typical delays include gaps between connecting to the server and sending its request, reading data from the server's response, or sending subsequent requests in a persistent connection. Disk-related blocking occurs when locating files on disk, or when reading file data before sending it to the client. Of the two, network blocking is more problematic, because client may delay indefinitely, while modern disk access typically takes less than 10ms.

The multi-process servers are conceptually the simplest, and are the oldest architectures for Web servers. One process opens the socket used to accept incoming requests, and then creates multiple copies of itself using the fork() system call. The earliest servers would fork a new process that exited after each request, but this approach quickly changed to a pool of pre-forked processes that serve multiple requests before exiting. On Unix-like systems, this model is the only option for Apache version 1, and the default for version 2.

Multi-threaded and event-driven servers use a single address space to improve performance and scalability, but also increase programming complexity. Sharing data in one address space simplifies bookkeeping, cross-request policy implementations, and application-level caching. The trade-off is programmer effort – multi-threaded programs require proper synchronization, and event-driven programs require breaking code into non-blocking portions. Both activities require more programmer effort and skill than simply forking processes.

While these architectures differ in memory consumption, scalability, and performance, well-written systems using any of these architectures can handle large volumes of traffic, enough for the vast majority of sites. A site's choice of web server likely depends on factors other than raw capacity, such as specific features, flexibility, operating system support, administrator familiarity, etc.

3 Design

Using a pipe-like mechanism and a simple API, Connection Conditioning performs application-level interposition on connection-related system calls, with all policy decisions made in user-level processes called filters. Applying the pipe design philosophy, these filters each perform sim-

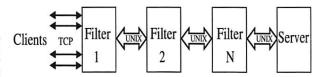


Figure 1: Typical Connection Conditioning usage – the server process invokes a series of filters connected to each other and the server via Unix-domain sockets. The first filter creates the actual TCP listen socket that is exposed to the clients. Clients connections are accepted at this filter, and are passed via file-descriptor passing through the other filters and finally to the server process.

ple tasks, but their combination yields power and flexibility. In this section, we describe the design of Connection Conditioning and discuss its impact on applications.

3.1 General Overview

Connection Conditioning replaces the server's code that accepts new connections, and interposes one or more filters. This design, shown in Figure 1, connects the filters and the server process using Unix-domain sockets. The TCP listen socket, used to accept new connections from clients, is moved from the server to the first filter. If we replaced the clients with standard input, this diagram would look like a piped set of processes spawned by a shell.

While modeled on Unix pipes, Connection Conditioning differs in several domain-specific respects. The most important difference is that rather than passing byte streams, the interface between filters as well as between the filter and server process is passing an atomic, delimited bundle consisting of the file descriptor (socket) and associated request. Using Unix-domain sockets allows open file descriptors to be passed from one process to another via the sendmsg() system call. While requests are passed between filters, the server's reply is written directly to the client socket.

Passing the client's TCP connection, rather than proxying the data, provides several benefits. First, the standard networking calls behave as expected since any calls that manipulate socket behavior or query socket information operate on the actual client socket instead of a loopback socket. Second, latency is also lower than a proxy-based solution, since data copying is reduced and the chance of any filter blocking does not affect data sent from the server to the client. Third, performance is also less impaired, since no extra context switches or system calls are needed for the response path, which transfer more data than the request path. Finally, the effort for using CC with existing server software is minimized, since all of the places where the server writes data back to the client are unaffected. Also unmodified are systems like external CGI applications, to which the server can freely pass the client's socket, just as it would without CC.

This approach allows filters to be much simpler than servers, and to be written in different styles – all of the parsing and concurrency management normally associated with accepting requests can be isolated into a single protocol-specific filter that is usable across many servers. Removing this complexity allows each filter and the server to use whatever architecture is appropriate. Programmers can use threads, processes, or events as they see fit, both in the server and in the filters. For simple servers and filters, it is even plausible to not even have any concurrency and handle only one request at a time, as we demonstrate later in the paper. This approach is feasible with Connection Conditioning because all connection management can be moved into the filters.

Note that the filters are tied to the number of features, not the number of requests, so a server will have a small number of filters even if it has many simultaneous connections. In practice, we expect that most servers will use 4 filters. Filter 1 will manage connections and take steps to reduce the possibility of denial-of-service attacks based on exhausting the number of connections. Filter 2 will separate multiple requests on a single connection, and present them as multiple separate connections, in order to eliminate idle connections from using server resources. Filter 3 will perform protocol-specific checks to stop malformed requests, buffer overflows, and other security attacks. Filter 4 can perform whatever request prioritization policy the server desires.

Filters are generally tied to the protocol, not the application, allowing filters to be used across servers, and encouraging "best practices" filters that consolidate protocol-related handling so that many servers benefit from historical information. For example, the "beck" denial-of-service attack [26] exploits a quadratic algorithm in URL parsing, and was discovered and fixed on Apache in 1998. The exact same attack is still effective against the thttpd server [1], despite being demonstrated in a thttpd-derived server in 2002 [21]. The beck attack is worse for thttpd than Apache, since thttpd is eventdriven, and the attack will delay all simultaneous connections, instead of just one process. Thttpd is used at a number of high-profile sites, including Kmart, Napster, MTV, Drudge Report, and Paypal. Using CC, a single security filter could be used to protect a range of servers from attacks, giving server developers more time to respond.

CC filters are best suited to environments that consist of request/response pairs, where no hidden state is maintained across requests, and where each transaction is a single request and response. In this scenario, all request-related blocking is isolated in the first (client-facing) filter, which only passes it once the full request has arrived. Intermediate filters see only complete requests, and do not have to be designed to handle blocking. If the server's responses can fit into the outbound socket buffer, then any

remaining blocking in the server may be entirely bounded and predictable. In these cases, the server can even handle just a single request at a time, without any parallelism. All of the normal sources of unpredictable blocking (waiting on the request, sending the reply) are handled either by CC filters or by the kernel. This situation may be very common in sensor-style servers with small replies.

To handle other models of connection operation, like persistent connections, the semantics of filters can be extended in protocol-specific ways. Since persistent connections allow multiple requests and responses over a single connection, simply passing the initial request to the server does not prevent all future blocking. After the first request is handled, the server may have to wait for further requests. Even if the server is designed to tolerate blocking, it may cause resources, such as processes or threads, to be devoted to the connection. In this case, the server can indicate to the filter that it wants the file descriptor passed to it again on future requests. Since the filter also has the file descriptor open, the server can safely close it without disconnecting the client. In this manner, the client sees the benefits of persistent connections, but the server does not have to waste resources managing the connection during the times between requests.

3.2 Connection Conditioning Library

To implement Connection Conditioning, we provide a library, shown in Figure 2. One function replaces the three system calls needed to create a standard TCP listen socket, and the rest are one-to-one analogues of standard Unix system calls. The parameters for most calls are identical to their standard counterparts, and the remaining parameters are instantly recognizable to server developers. We believe that modifying existing servers to use Connection Conditioning is straightforward, and that using them for new servers is simple. Any of these calls can be used in process-based, threaded, or event-driven systems, so this library is portable across programming styles. This library also depends on only standard Unix system calls, and does not use any kernel modifications, so is portable across many operating systems. The library contains 244 lines of code and 89 semicolons. Its functions are:

cc_createlsock — instantiates all of the Connection Conditioning filters used by this server. Each filter in the NULL-terminated array filters[] is spawned as a separate process, using any arguments provided by the server. Each filter shares a Unix-domain socket with its parent. The list of remaining filters to spawn is passed to the newly-created filter. The final filter in the list creates the listen socket that accepts connections and requests from the client. The server specifies all of the filters, as well as the parameters (address, port number, backlog) for the listen socket, in the cc_createlsock call. The server process no longer needs to call the

```
int cc_createlsock(struct in_addr sin_addr,
                   in_port_t sin_port,
                   int backlog,
                   char *filters[]);
int cc_accept(int s, struct sockaddr *addr,
              socklen t *addrlen);
ssize_t cc_read(int fd, void *buf,
                size_t count);
int cc_close(int fd, int closeAllFilters);
int cc_select(int n, fd_set *readfds,
              fd_set *writefds,
              fd_set *exceptfds,
              struct timeval *timeout);
int cc_poll(struct pollfd fds[],
            nfds_t nfds, int timeout);
int cc_dup(int oldfd);
int cc_dup2(int oldfd, int newfd);
pid_t cc_fork(void);
```

Figure 2: Connection Conditioning Library API

socket/bind/listen system calls itself. The return value of cc_createlsock is a socket, suitable for use with cc_accept. Our filter instantiation differs from Unix pipes, since the server instantiates them, instead of having the shell perform the setup. This approach requires much less modification for existing servers, and it also avoids conflicts with stdin/stdout.

cc_accept - this call replaces the accept system call, and behaves similarly. However, instead of receiving the file descriptor from the networking layer, it is received from the filter closest to the server. The file descriptor still connects to the client and is passed using the sendmsg() system call, which also allows passing the request itself. The request is read and buffered, but not presented yet.

cc_read - when cc_read is first called on a socket from cc_accept, it returns the buffered request, and behaves as a standard read system call on subsequent calls. The reason for this behavior is because the socket is actually terminated at the client. If any filter were to write data into the socket, it would be sent to the client. So, the filters send the (possibly updated) request via sendmsg when the client socket is being passed.

In multi-process servers, with many processes sharing the same listen socket, the atomicity of sendmsg and recvmsg ensures that the same process gets both the file descriptor and the request. If requests will be larger than the Unix-domain atomicity limit, each process has its own Unix-domain socket to the upstream filter, and calling cc_accept sends a sentinel byte upstream. The upstream filter sends ready requests to any willing downstream filter on its own socket.

cc_close – since the same client socket is passed to all of the filters and the actual server, some mechanism is needed to determine when the socket is no longer useful. Some filters may want to keep the connection open longer than the server, while other filters may not care about the connection after passing it on. The cc_close call provides for this behavior – the server indicates whether only it is done with the connection or whether it and all filters should abandon the connection. The former case is useful for presenting multiple requests on a persistent connection as multiple separate connections. The latter case handles all other scenarios, as well as error conditions where a persistent connection needs to be forcibly closed by the server.

cc_select, cc_poll — these functions are needed by event-driven servers, and stem from transferring the request during cc_accept. Since the request is read and buffered by the CC library, the actual client socket will have no data waiting to be read. Some event-driven servers optimistically read from the socket after accept, but others use poll/select to determine when the request is ready. In this case, the standard system calls will not know about the buffered request. So, we provide cc_select and cc_poll that check the CC library's buffers first, and return immediately if buffered requests exist. Otherwise, they simply call the appropriate system calls.

cc_dup, cc_dup2, cc_fork – These functions replace the Unix system calls dup, dup2, and fork. All of these functions affect file descriptors, some of which may have been created via cc_accept. As such, the library needs to know when multiple copies of these descriptors exist, in order to adjust reference counts and close them only when the descriptor is closed by all readers.

While the CC Library functions are easily mapped to standard system calls, transparently converting applications by replacing dynamically-linked libraries is not entirely straightforward. The cc_createlsock call replaces socket, bind, and listen, but these calls are also used in other contexts. Determining future intent at the time of the socket call may be difficult in general.

4 Evaluation

Our evaluation of Connection Conditioning explores three issues: writing servers, CC performance, and CC security. We also examine filter writing, but this issue is secondary to developers if the filters are reusable and easily extensible. We first present a simple server designed with Connection Conditioning in mind, and then discuss the effort involved in writing filters. We compare its performance to other servers, and then compare the performance effects of other filters. Finally, we examine various security scenarios, and show that Connection Conditioning can improve server security.

4.1 A Simple Server

To demonstrate the simplicity of writing a Web server using Connection Conditioning, we build an extremely simple Web server, called CCServer. Using this server, we test whether the performance of such a simple system would be sufficient for most sites. The pseudo-code for the main loop, almost half the server, is shown in Figure 3. This listing, only marginally simplified from the actual source code, demonstrates how simple it can be to build servers using Connection Conditioning. The total source for this server is 236 lines, of which 80 are semicoloncontaining lines. In comparison, Flash's static content handling and Haboob (not including NBIO) require over 2500 semicolon-lines and Apache's core alone (no modules) contains over 6000. Note that we are not advocating replacing other servers with CCServer, since we believe it makes sense to simply modify servers to use CC.

CCServer's design sacrifices some performance for simplicity, and achieves fairly good performance without much effort. Its simplicity stems from using CC filters, and avoiding performance techniques like applicationlevel caching. CCServer radically departs from current server architectures by handling only one request at a time. The only exception is when the response exceeds the size of the socket buffer, in which case CCServer forks a copy of itself to handle that request. Within limits, the socket buffer size can be increased if very popular files are larger than the default, in which case one time cache miss in the main process is also justified - with the use of the zero-copy sendfile call, multiple requests for a file consume very little additional memory beyond the file's data in the filesystem cache. Parallelism is implicitly achieved inside the network layer, which handles sending the buffered responses to all clients.

CCServer ignores disk blocking for two reasons: decreasing memory costs means that even a cheap system can cache a reasonably large working set, and consumergrade disk drives now have sub-10ms access times, so even a disk-bound workload with small files can still generate a fair amount of throughput. To really exceed the size of main memory, the clients must request fairly large files, which can be read from disk with high bandwidth. It is possible to build degenerate workloads with thousands of small-file accesses, but using a filter that gives low priority to heavy requestors (described in Section 4.2) will limit the performance degradation that other clients see.

The only obvious denial-of-service attack we can see in this approach is that an attacker could request many large files, causing a large number of processes to exist, and could make the situation worse by reading the response data very slowly. This situation is not unique to CCServer – any server, particularly threaded or processoriented servers, are vulnerable to these attacks. All

```
char *filters[] = {"flt prior", "flt_persist",
                   "flt_request", NULL);
char request[MAXREQUEST];
int s. c:
s = cc_createlsock(INADDR_ANY, SERV_PORT,
                   BACKLOG, filters);
while ((c = cc_accept(s, NULL, NULL)) >= 0) {
 bool is_child = false, send_body = true;
  off_t offset
                 = 0;
 fileinfo file;
  cc_read(c, request, sizeof(request));
  file = parse_and_openfile(request);
  send_header(c, file.size);
  set_sendbufsize(c, SENDBUFSIZE);
  if (file.size > SENDBUFSIZE) {
     /* let a child process send the body */
      if (cc_fork() != 0) send_body = false;
      else is_child = true;
  if (send_body) /* send the body */
    sendfile(c, file.fd, &offset, file.size);
  cc close(c);
  close(file.fd);
  if (is_child) return 0;
```

Figure 3: Pseudo code of the really simple CCServer

of these techniques can be handled similar to how they would be handled in other servers. We could set process limits in the shell before launching the server, in order to ensure that too many processes are not created. To handle the "slow reading" attack, we could split the sendfile into many small pieces, and exit if any piece is received too slowly. With CC filters, we could use a filter that places low priority on heavy requestors, which would reduce the priority of any attacker.

All of the other concerns that one would expect, such as how long to wait between a connection establishment and the request arrival, how long to keep persistent connections open, etc., are handled by filters outside the server. Normally, all of these issues would cause a server that handled only one request at a time to block for unbounded amounts of time, and would necessitate some parallelism in the server's architecture, even for simple/short requests.

4.2 Filters

We have developed filters that implement different connection management and security policies. We find filter development relatively straightforward, and that the basic filter framework is easy to modify for different purposes. Common idioms also emerge in this approach, leading us to believe that filter development will be manageable for the programmers who need to write their own.

	Total (Semicolons)
Packaging	687 (248)
Persistence	+76 (+26)
Priority	531 (211)
Slow Read	587 (212)

Table 1: Line counts for filters – the persistence filter is conditionally-compiled support in the packaging filter, so its counts are shown as the extra code for this feature. The other filter line counts include the basic framework, which is 413 lines of source, and 152 semicolons.

We have found two common behavioral styles for filters, and these shape their design. Those that implement some action on individual requests, such as stripping pathname components or checking for various errors, can be designed as a simple loop that accepts one request, processes it, and passes it to the next filter. Those that make policy decisions across multiple requests are conceptually small servers themselves.

These filters are an important aspect of the system, since they are key to preserving programming style while enhancing security. In traditional multi-process servers without Connection Conditioning, making a policy decision across all active requests is difficult enough, but it is virtually impossible to consider those requests that are still waiting in the accept queue. Since the number of those requests may exceed the number of processes in the server, certain security-related policy decisions are unavailable to these servers.

The filters, in contrast, can use a different programming style, like event-driven programming, so that each request consumes only a file descriptor instead of an entire process. In this manner, the filters can examine many more requests, and can more cheaply make policy decisions. We use a very simple event-driven framework for the policy filters, since we are particularly interested in trying to implement policies that can effectively handle various DoS attacks. To gain cross-platform portability and efficiency, we use the libevent library [19], which supports platform-specific event-delivery approaches (kqueue [15], epoll, /dev/poll) in addition to standard select and poll. Our filters include:

Request packaging – this filter is often the first filter in any server. It accepts connections, reads the requests, and hands complete requests to the next filter. By making the filter event-driven, it can handle attacks that try to starve the server by opening thousands of connections without sending requests. The filter is only limited by the number of file descriptors available to it, and we implement some simple policies to prevent starvation. Any connection that is incomplete (has not sent a full request) before a configurable timeout is terminated, and if the filter is running

out of sockets, incomplete connections are terminated by network address. This filter maintains a 16-ary tree organized by network address, where each node has a count of all open connections in its children. The filter follows the path with the heaviest weights, ensuring that the connection it terminates is coming from the range of network addresses with the most incomplete connections.

Persistent connection management – while persistent connections help clients, they present connection management problems for servers, so this filter takes multiple requests from a persistent connection and separates them into individual requests. When the server is done with the current request, it closes the connection, and this filter re-sends the next request as a new connection. Since the filters keeps a socket open, the server closing a persistent connection is only a local operation, and is not visible to the client. We expect that this filter would be the second filter after the request packaging filter.

Recency-based prioritization – this filter acts as a holding area after the full request has arrived. It provides a default policy that makes high-rate attacks less effective, without requiring any feedback or throttling information from the server. As a side-effect, it also provides simple fairness among different users. This would be the last filter before the server. This filter basically accepts all requests coming from the previous filter, and then picks the highest-priority request when the server asks for one. The details of this approach are described in Section 4.2.1.

Slow read prevention – this filter limits the damage caused by "slow read" clients, who request a large file and then keep the connection open by reading the data slowly. In a DoS scenario, if a client could keep the connections open arbitrarily long, the prioritization filter alone would not prevent it from having too many connections. This filter explicitly checks how many concurrent connections each client has, and delays or rejects requests from any client range that is too high. We currently set the defaults to allowing no more than 25% of all connections from a /8 network range, 15% from a /16, and 10% from a /24. This approach limits slow-read DoS, but can not fully protect against DDoS. Still, any security improvement is a benefit for a wide range of servers.

We have also developed other more specialized filters, such as ones that look for oddly-formatted requests, detect and strip the beck attack, etc. Line count information for the filters described above are presented in Table 1.

4.2.1 The Recency Algorithm

To handle rate-based attacks coming from sets of addresses, we use an algorithm that aggregates traffic statistics automatically at multiple granularities, but does not lose preciseness. We break the network address into 8 pieces of 4 bits each. We use an 8 deep 16-ary tree, with

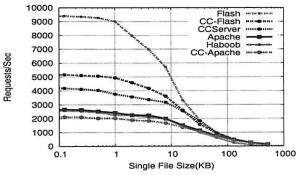


Figure 4: Single File Requests/sec

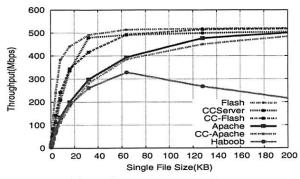


Figure 5: Single File Throughput

an LRU list maintained across the 16 elements of each parent. Each node also contains a count to indicate how many requests are stored in its subtrees. The tree is lazily allocated – any levels are allocated only when distinct addresses exist in the subtree. When a new request arrives, it is stored in the tree, creating any nodes that are needed, and updating all counts of requests. When it is time to provide the next request to be serviced, we descend each level of the tree, using the LRU child with a nonzero request count at each level. The request chosen by this process is removed, all counts are updated, and all children along the path are moved to the ends of their respective LRU lists. Subtrees without requests can be pruned if needed.

If an attacker owns an entire range of network addresses, a low-frequency client from another address range will always take priority in having its requests serviced or its incomplete connections kept alive. Even if the low-frequency client is more active than any individual compromised machine, this algorithm will still give it priority due to the traffic aggregation behavior. At the same time, the aggregation does not lose precision – if even a single machine in the attacker's range remains uncompromised, when it does send requests, they will receive priority over the rest of the machines in that range.

4.3 Performance Evaluation

Though performance is not a goal of Connection Conditioning, we evaluate it so that designers and implementors have some idea of what to expect. While we believe it is true that performance is generally not a significant factor in these decisions, it would become worrying if the performance impact caused any significant number of sites to reject such an approach. As we show in this section, we believe that the performance impact of Connection Conditioning is acceptable.

4.3.1 Testbed and Servers

Our testbed servers consist of a low-end, single processor desktop machine, as well as an entry-level dual-core server machine. Most of our tests are run on a \$200 Microtel PC from Wal-Mart, which comes with a 1.5 GHz AMD Sempron processor, 40 GB IDE hard drive, and a

built-in 100 Mbps Ethernet card. We add 1 GB of memory and an Intel Pro/1000 MT Server Gigabit Ethernet network adapter, bringing the total cost to \$396. Using a Gigabit adapter allows us to break the 100 Mbps barrier, just for the sake of measurement. The dual-core server is an HP DL320 with a 2.8 GHz Intel Pentium D 820, 2 GB of memory, and a 160GB IDE hard drive. This machine is still modestly priced, with a list price of less than \$3000. Both machines run the Linux 2.6.9 kernel using the Fedora Core 3 installation. Our test harness consists of four 1.5 GHz Sempron machines, connected to the server via a Netgear Gigabit Ethernet switch.

In various places in the evaluation, we compare different servers, so we briefly describe them here. We run the Apache server [3] version 1.3.31, tuned for performance. Where specified, we run it with either the default number of processes or with higher values, up to both the "soft limit," which does not require recompiling the server, and above the soft limit. The Flash server [18] is an event-driven research server that uses select to multiplex client connections. We use the standard version of it, rather than the more recent version [23] that uses sendfile and epoll. The Haboob server [32] uses a combination of events and threads with the SEDA framework in Java. We tune it for higher performance by increasing the filesystem cache size from 200MB to 400MB. CCServer is our simple single-request server using Connection Conditioning. CC-Apache and CC-Flash are the Apache and Flash servers modified to use Connection Conditioning. In all of the servers using Connection Conditioning, we employ a single filter unless otherwise specified. Since the CC Library currently only supports C and C++, we do not modify Haboob. All servers have logging disabled since their logging overheads vary significantly.

4.3.2 Single-File Tests

The simplest test we can perform is a file transfer microbenchmark, where all of the clients request the same file repeatedly in a tight loop. This test is designed to give an upper bound on performance for each file size, rather than being representative of standard traffic. The results of

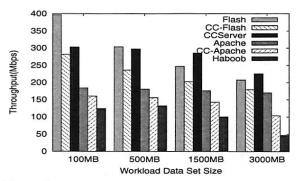


Figure 6: SpecWeb99-like workloads on the Microtel machine

this test on the Microtel machine are shown in Figures 4 and 5 for request rate and throughput, respectively. The relative positions of Flash, Apache, and Haboob are not surprising given other published studies on their performance. Performance on the HP server is higher, but qualitatively similar, and is omitted for space.

The performance of CCServer is encouraging, since this would mean that it should have acceptable performance for any site using Apache. Any performance loss due to forking overhead once the response size exceeds the socket buffer size is not particularly visible. This server is clearly not functionally comparable to Apache, but given the use of multiple processes in request handling, we are pleased with the results.

Using Connection Conditioning filters with other servers also seems promising, as seen in the results for CC-Apache and CC-Flash. Both show performance loss when compared to their native counterparts, but the loss is more than likely tolerable for most sites. We investigate this further on a more realistic workload mix next.

4.3.3 More Realistic Workloads

While the single-file tests show relative request processing costs, they do not have the variety of files, with different sizes and frequency distributions, that might be expected in normal Web traffic. For this, we also evaluate these servers using a more realistic workload. In particular, we use a distribution modeled on the static file portion of SpecWeb99 [28], which has also been used by other researchers [23, 30, 32]. The SpecWeb99 benchmark scales data set size with throughput, and reports a single metric, the number of simultaneous connections supported at a specified quality of service.

We instead use fixed data set sizes and report the maximum throughput achieved, which provides a broader range of results for each server. We maintain the general access patterns – a data set contains a specified number of files per directory, with a specified access frequency for files within each directory. The access frequency of the directories follows a Zipf distribution, so the first directory is accessed N times more than the N^{th} directory.

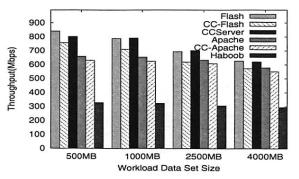


Figure 7: SpecWeb99-like workloads on the HP server

The results of these capacity tests, shown in Figures 6 and 7, show some expected trends, as well as some subtler results. The most obvious trend is that once the data set size exceeds the physical memory of the machine, the overall performance drops due to disk accesses. For most servers, the performance prior to this point is roughly similar across different working set sizes, indicating very little additional work is generated for handling more files, as long as the files fit in memory. CCServer performs almost three times as well on the HP server as the Microtel box, demonstrating good scalability with faster hardware.

The performance drop at 3 or 4 GB instead of 1.5/2.5 GB can be understood by taking into account SpecWeb's Zipf behavior. Even though a 1.5 and 2.5 GB data sets exceed the physical memories of the machines, the Zipf-distributed file access causes the more heavily-used portion to fit in main memory, so this size has a mix of inmemory and disk-bound requests. At 3GB, more requests are disk-bound, causing the drop in performance across all servers. The HP machine, with a larger gap between CPU speed and disk speed, shows relatively faster degradation with the 4GB data set. Though CCServer makes no attempt to avoid disk blocking, its performance is still good on this workload.

In general, the results for the CC-enabled servers are quite positive, since their absolute performance is quite good, and they show less overhead than the single file microbenchmarks would have suggested. The main reason for this is that the microbenchmarks show a very optimistic view of server performance, so any additional overheads appear to be much larger. On real workloads, the additional data makes the overall workload less amenable to caching in the processor, so the overheads of Connection Conditioning are less noticeable.

4.3.4 Chained CC Filters

Inter-process communication using sockets has traditionally been viewed as heavyweight, which may raise concerns about the practicality of using smaller, single-purpose filters chained together to compose behavior.

To test the latency effects, we vary the number of filters

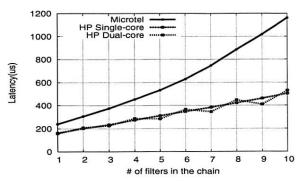
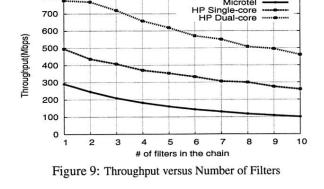


Figure 8: Latency versus Number of Filters



800

used in CCServer, and have a single client issue one request at a time for a single 100-byte file. All of the filters except the first are dummy filters, simply passing along the request to the next filter. These results, shown in Figure 8, show that latency is nearly linear in the number of filters, and that each filter only adds 34 μ seconds (HP) or 94 μ seconds (Microtel) of overhead. Compared to widearea delays of 100ms or more, the overhead of chained filters should not be significant for most sites.

The performance effects of chained filters are shown in Figure 9, where an in-memory SpecWeb-like workload is used to drive the test. Given the near-linear effect of multiple filters on latency, the shape of the throughput curve is not surprising. For small numbers of filters, the decrease is close to linear, but the degradation slows down as more filters are added. Even with what we would consider to be far more filters than most sites would use, the throughput is still well above what most sites need.

CCServer performs better on the HP server than the Microtel box on both tests, presumably due to the faster processor coupled with its 1MB L3 cache. The dual-core throughputs scale well versus the single-core, indicating the ability of the various filters in the CC chain to take advantage of the parallel resources. While enabling both cores improves the throughput in this test, it does not improve the latency benchmark, because only one request progresses through the system at a time. The sawtooth pattern stems from several factors: some exploitable parallelism between the clean-up actions of one stage and the start-up actions of the next, SMP kernel overhead, and dirty cache lines ping-ponging between the two independent caches as filters run on different cores.

4.4 Security Evaluation

Here we evaluate the security effects of Connection Conditioning, particularly the policy filters we described in Section 4.2. Note that some of these tests have been used in previous research [21] – our contribution is the mechanism of defending against them, rather than the attacks.

Our primary reason for selecting these tests is that they are simple but effective – they could disrupt or deny ser-

vice to a large fraction of Web sites, and they do not require any significant skill. Each attacking script requires less than 200 lines of code and only a cursory knowledge of network programming and HTTP protocol mechanics. Some of these attacks would also be hard to detect from a traffic viewpoint – they either require very little bandwidth, or their request behavior can be made to look like normal traffic. We focus on the Apache server both because its popularity makes it an attractive target, and because its architecture would normally make some security policies harder to implement. All results are shown for only the Microtel box because these tests focus primarily on qualitative behavior.

4.4.1 Starvation Test - Incomplete Connections

To measure the effect of incomplete connections on the various servers, we have one client machine send a stream of requests for small files, while others open connections without sending requests. We measure the traffic that can be generated by the regular client in the presence of various numbers of incomplete connections. These results are shown in Figure 10, and show various behavior for the different servers. For the process-based Apache server, each connection consumes one process for its life. We see that a default Apache configuration takes only 150 connections, at which point performance drops. Apache employs a policy of waiting 300 seconds before terminating a connection, so at this limit, throughput drops to 0.5 requests/second. Though Flash and Haboob are eventdriven, neither have support for detecting or handling this condition. Flash's performance slowly degrades with the number of incomplete connections, and becomes unusable at 32K connections, while Haboob's performance sharply drops after 100 incomplete connections. Flash's performance degradation stems from the overheads of the select system call [4].

With the CC Filters, all of these servers remain operational under this load, even with 32K incomplete connections. Since the filter terminates the oldest incomplete connection when new traffic arrives, it can still handle workloads of 1800 requests/sec for CC-Apache, and

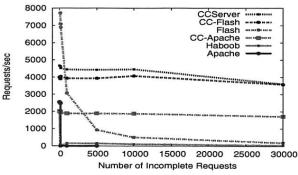


Figure 10: Number of Incomplete Connections Handled

3700 requests/sec for CCServer and CC-Flash. This test demonstrates the architecture-neutral security enhancement that Connection Conditioning can provide – both a multi-process server and an event-driven server handle this attack better with Connection Conditioning than their own implementation provides.

4.4.2 Prioritization Test

Though the request packaging filter closes connections in a fair manner, the previous test does not demonstrate fairness for valid requests, so we devise another test to measure this effect. The test consists of a number of clients requesting large files from a default Apache, which can handle 150 simultaneous requests. The remaining requests are queued for delivery, so an infrequent client may often find itself waiting behind 150 or more requests. The infrequent client in our tests requests a small file, to observe the impact on latency.

The results of this test, in Figure 11, show the effect on latency of the infrequently-accessing client. The latency of the small file fetch is shown as a function of the number of clients requesting large files. Without the prioritization filter, Apache treats the request in roughly first-come, first-served order. When the total number of clients is less than the number of processes, the infrequent client can still get service reasonably quickly. However, once the number of clients exceeds the number of server processes, the latency for the infrequent client also increases as more clients request files.

With CC-Apache and the prioritization filter, though, the behavior is quite different. The increase in the number of large-file clients leads to a slight increase in latency once all of the processes are busy. After that point, the latency levels again. This small step is caused by the infrequent client being blocked behind the next request to finish. Once any request finishes, it gets to run, so the latency stays low.

Performing this kind of prioritization in a multipleprocess server would be difficult, since each connection would be tied to a process. As a result, it would be hard for the server to determine what request to handle next. With

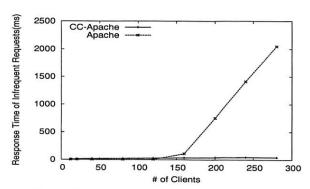


Figure 11: Latency versus Number of Other Clients

Connection Conditioning, the filter's policy can view all outstanding requests, and make decisions before the requests reach the server.

4.4.3 Persistence Test

Persistent connections present another avenue for connection-based starvation, similar to the incomplete connection attack. In this scenario, an attacker requests a persistent connection, requests a small file, and keeps the connection open. To avoid complete starvation, any reasonable production-class server will have some mechanism to shut down such connections either after some timeout or under file descriptor pressure.

Implementing a self-managing solution is tied to server architecture, complicating matters. While detecting file descriptor pressure is cheap in event-driven servers, they are also less vulnerable, since they can utilize tens of thousands of file descriptors. In contrast, multi-process servers are designed to handle far fewer simultaneous connections, and determining that persistent connection pressure exists requires more synchronization and inter-process communication, reducing performance. The simplest option in these circumstances is to provide administrator-controlled configuration options regarding persistent connection behavior as Apache does. However, the trade-off is that if these timeouts are too short, they make persistent connections less useful, while if they are too long, the possibility of running out of server processes increases.

Figure 12 shows the effect of an attacker trying to starve the server via persistent connections. We use Apache's default persistent connection timeouts of 15 seconds and 150 server processes. An attacker opens multiple connections, requests small files, and holds the connection open until the server closes it. For any closed connection, the attacker opens a new connection and repeats the process. We vary the number of connections used by the attacker. We also have 16 clients on one machine requesting the SpecWeb99-like workload with a 500 MB data set size. We show the throughput received by the regular clients as a function of the number of slow persistent connections. On Apache, the throughput drops beyond 150 persistent

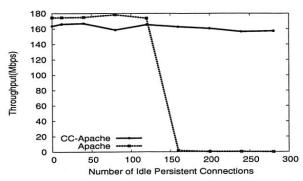


Figure 12: Throughput under Persistent Connection Limits

connections, but CC-Apache shows virtually no performance loss. Its maximum performance is lower than standard Apache due to the CC filters, but it supports more open connections. Apache's server processes never see the waiting periods between requests. This support only required modifying 8 lines in Apache.

5 Discussion

In this section, we discuss some alternatives to Connection Conditioning, some of the objections that may be raised to our claims, and possible deployment questions.

5.1 Novelty and Simplicity

Our contribution in Connection Conditioning is the observation that Unix pipes can be applied to servers, providing all of the benefits associated with text processing (simplicity, composability, and separation of concerns) while still providing adequate performance. In retrospect, this may seem obvious, but we believe that Connection Conditioning's design and focus on adoptability are directly responsible for its other benefits. Our approach allows vastly simpler servers with performance that approaches or even exceeds the designs introduced in the past few years. Particularly for small servers, such as sensors, our approach provides easy development with a broad range of protection, something not available in other approaches. We make no apologies for building on the idea of Unix pipes given the option to build on a great idea, we see no reason to develop new approaches purely for the sake of novelty.

CC also provides the ability to incorporate best practices into existing servers, without having to start from a clean slate. Given the state of today's hardware, someone designing a server from scratch may develop a design similar to Connection Conditioning. However, even many research servers, with no compatibility constraints, have become increasingly complicated over time, rather than simpler. We consider the ability to support existing servers like Apache while still allowing new designs like CCServer to be a contribution of this work.

5.2 Rich Web Server APIs

Several servers provide rich APIs that can be used to inspect and modify requests and responses - Apache has its module format, Microsoft developed ISAPI, Netscape developed NSAPI, and Network Appliance developed ICAP. Any of these could be used to protect their host server from attacks like the beck attack mentioned earlier. However, we believe Connection Conditioning can provide protection from a separate class of attacks not amenable to protection via server APIs. These attacks, such as the "incomplete connection" starvation attack, waste server resources as soon as the connection has been accepted, and these connections are accepted within the framework of the server. Particularly for process-based servers, the resources consumed just by accepting the connection can be significant. By moving all of the inspection and modification outside the server, Connection Conditioning provides protection against this class of attack. Even event-driven servers can expend more state than Connection Conditioning - in our request prioritization example, we may want to select from tens of thousands of possible connections, particularly when we are under attack. The richness of the server's internal API does no good in this kind of example, since the server may not even be able to accept all of the connections without succumbing to the attack.

Some of CC's other benefits, such as relieving the server of the work of maintaining persistent connections, cannot be done inside all servers without architectural changes. The persistent connection attack we have shown is particularly effective, since regular servers would have to have global knowledge of the state of all requests in order to detect it. With CC, no server re-architecting is required, since this work can be done easily in the filters.

5.3 Security

We have seen that CC protects servers against several DoS attacks, and that it enables other types of protocolspecific security filters. Given how little bandwidth some of these attacks require, and given Apache's wide deployment, we feel that CC can provide an immediate practical security benefit. From a design standpoint, using CC with filters can also provide other benefits - privileged operations, such as communicating with authentication servers or databases, can be restricted to specific policy filters, moving sensitive code out of the larger code base of the main server. These filters, if designed for re-use, can also be implemented using best practices, and can be more thoroughly tested since wider deployment and use with multiple servers is more likely to expose security holes. We admit that some of these benefits will be hard to quantify, but we also feel that some of them are self-evident – moving code out of a large, monolithic server code base and executing it in a separate address space is likely to restrict the scope of any security problem.

5.4 Scope

While our evaluation of Connection Conditioning has focused mostly on Web servers, we believe CC has a fairly broad scope - it is suitable to many request/reply environments that tend to have relatively short-duration "active" periods of their transactions. Our focus on Web servers is mostly due to pragmatism - Web servers are widely deployed, and they provide ample opportunities for comparisons, so our evaluation of CC can be independently assessed. In addition to the server protection offered by CC, we also hope to use it in developing lightweight, DoSresistant sensors for PlanetLab. We run several sensors on PlanetLab for providing status information - CoMon provides node-level information, such as CPU load and disk activity, while CoTop provides account-level (slice-level) information, such as number of processes and memory consumption. While these tools all use HTTP as a transport protocol, they are not traditional Web servers. By using CC for these tools, we can make them much more robust while eliminating most of their redundant code.

CC is not suitable for all environments, and any server with very long-lived transactions may not gain simplicity benefits from it. Video server match this profile, where a large number of clients may be continuously streaming data over long-lived connections for an hour or more. In this case, CC is no better than other architectures at providing connection management. In all likelihood, this case will require some form of event-driven multiplexing at the server level, whether it is exposed to the programmer or not. CC can still provide some filtering of requests and admission control, but may not be a significant advantage in these scenarios. This example is distinct from the persistent connection example we provided earlier - the difference is that with persistent connections, the longlived connections may be handling a number of shortlived transfers. In that case, CC can reduce the number of connections actually being handled by the server core.

6 Related Work

While this paper has argued that performance-related advances in server design are of marginal benefit to most Web sites, some classes of servers do see benefit from many advances. Banga and Mogul improved the *select()* system call's performance by reducing the delay of finding ready sockets [5]. They subsequently proposed a more scalable alternative system call [7], which appears to have motivated *kqueue()* on BSD [15] and *epoll()* on Linux. Caching Web proxy servers have directly benefited from this work, since they are often in the path of every request from a company or ISP to the rest of the Internet. Any mechanism that reduces server latency is desirable in these settings. Examining the results from the most recent Proxy Cache-off [22] suggests that vendors are in fact interested in more aggressive server designs. In these

environments, CC may not be the best choice, but many ISPs still use the low-performance Squid proxy, so CC's overhead may be quite tolerable in these environments.

The method of filters we present is very general and allows customizable behavior. The closest approach we have found in any other system is the "accept filter" in FreeBSD, which provides an in-kernel filter with a hardcoded policy for determining when HTTP requests are complete. However, it must be specifically compiled into the kernel or loaded by a superuser. This approach resulted in opening the possibility of denial-of-service attacks on the filter's request parsing policy [10], which would have prevented the application from processing any requests. It would also be unable to handle some of the other starvation attacks we have covered in this paper. Similarly, IIS has an in-kernel component, the Software Web Cache, to handle static content in the kernel itself. While this approach can use kernel interfaces to improve scalability, its desirability may depend on whether the developer is willing to accept the associated risks of putting a full server into the kernel. For some of the cases we have discussed, such as developing simple, custom sensors that use HTTP as a transport protocol, in-kernel servers may provide little benefit if the infrastructure cannot be leveraged outside of its associated tasks.

Some of our security policies are shaped by work on making event-driven servers more responsive under malicious workloads [21]. We have attempted, as much as possible, to broaden these benefits to all servers, with as little server modification as possible. We believe that our recency-based algorithm is an elegant generalization of the approach presented in the earlier work.

While many of our evaluations have used Apache, both because of its popularity and because of the difficulty of performing certain security-related operations in a multiprocess server, we believe our approach is fairly general. We have shown that it can be applied to Flash, an event-driven Web server. We believe that it is broadly amenable to other designs, including hybrid thread/event designs such as Knot [30]. While we tried to demonstrate this feasibility, we were unable to get the standard Knot package working in the 2.6 or 2.4 Linux kernel. We believe Connection Conditioning would benefit a system like Knot most by preventing starvation-based attacks. The higher-performance version of Knot, Knot-C, uses a smaller number of threads to handle a large number of connections, possibly leaving it open to this kind of attack. In conjunction with CC Filters, only active connections would require threads in Knot.

Some work has been done on more complicated controllers for overload control [25, 31], which moves request management policy inside the server. If such an approach were desired in Connection Conditioning, it could be done via explicit communication between the filter and servers,

using shared memory or other IPC mechanisms. However, implementing such schemes as filters has the benefit of leaving the design style of the filter up to the developer, instead of having to conform to the server's architecture. Having the filter operate in advance of the server's accepting connections has the possibility of reducing wasted work. Servers would still be free to enforce whatever internal mechanisms they desired.

Similarly, resource containers [6] have been used to provide priority to classes of clients in event-driven and process-based servers. This mechanism can be used to provide a specified level of traffic to friendly clients even when malicious clients are generating heavy traffic. This approach depends on early demultiplexing in the kernel, and forcing policy decisions into the kernel to support this behavior. We believe that resource containers can be used in conjunction with Connection Conditioning, such that livelock-related policies are moved into the kernel with resource containers, and that the CC Filters handle the remaining behavior at application-level.

Finally, a large body of work exists on some form of interposition, often used for implementing flexible security policies. Some examples of this approach include Systrace [20], which can add policies to existing systems, Kernel Hypervisors [16], which can generalize the support for customizable, in-kernel system call monitoring, and Flask [27], an architecture designed to natively provide fine-grained control for a microkernel system. While some of CC's mechanisms could be implemented using system call interposition, the fundamental concerns of CC differ from these projects since filters in CC are trusted, and are logically extending the server, rather than viewing the server in an adversarial context. In this vein, CC is more similar to approaches like TESLA [24], that are designed to extend/offload the functionality of existing systems. Combining CC with TESLA, which provides session-layer services, would be a logical pairing, since their focus areas are complementary. The reason for not using some form of system call interposition in the current CC is that some decisions are simpler when made explicitly - for example, a purely interposition-based system may have a difficult time detecting all uses of the common networking idiom of socket/listen/accept, especially if other operations, such as fork () or dup (), are interleaved. Making CC calls explicit greatly simplifies the library.

7 Future Work

The next step for Connection Conditioning would be to add kernel support for the interposition mechanisms, while still keeping the server and filters in user space. We intentionally keep the filters in user space because we believe that the flexibility of having them easily customizable outweighs any performance gains of putting them in the kernel. We also believe that by moving only the mechanisms into the kernel, Connection Conditioning can be used without requiring root privilege.

The general idea is to allow the server to create its listen socket, and then have a minimal kernel mechanism that allows another process from the same user to "steal" any traffic to this socket. The first filter would then perform connection passing to other filters using the standard mechanisms. However, when the final filter wants to pass the connection to the server, it uses another kernel mechanism to re-inject the connection (file descriptor and request) where it would have gone to the server. In effect, the entire filter chain is interposed between the lower half of the kernel and the delivery to the server's listen socket. Such a scheme would be transparent to the server, and could operate without any server modification if the ability to split persistent connections into multiple connections is not needed. Otherwise, all of the other CC library functions could be eliminated, with only cc_close exposed via the API. Some extra-server process would have to launch all of the filters, and indicate which socket to steal, but this infrastructure is also minimal.

For closed-source servers where even minimal modifications are not possible, this approach may be the only mechanism to use Connection Conditioning. However, since our current focus is on experimentation, the librarybased approach provides three important benefits: it is portable across operating systems and kernel versions, it requires less trust from a developer wanting to experiment with it, and it is easier to change if we discover new idioms we want to support. At some point in the future, after we gain more experience with Connection Conditioning, we may revisit an in-kernel mechanism specifically to support closed-source servers.

8 Conclusions

While server software design continues to be an active area of research, we feel it is worthwhile to assess its chances for meaningful impact given the current state of hardware and networking. We believe that performance of most servers is good enough for most sites, and that advances in simplifying server software development and providing better security outweigh additional performance gains. We have shown that a design inspired by Unix pipes, called Connection Conditioning, can provide benefits in both areas, and can even be used with existing server software of various designs. While this approach has a performance impact, we have demonstrated that even on laughably cheap hardware, this system can handle far more bandwidth than most sites can afford.

Connection Conditioning provides these benefits in a simple, composable fashion without dictating programming style. We have demonstrated a new server that is radically simpler than most modern Web servers, and have shown that fairly simple, general-purpose filters can be used with this server and others to provide good performance and security. The current implementation runs entirely in user space, which gives it more flexibility and safety compared to a kernel-based approach. However, a kernel-space implementation of the mechanisms is possible, allowing for improved performance while retaining the flexibility of user-space policies.

Overall, we believe that Connection Conditioning holds promise for simplifying server design and improving security, and should be applicable to a wide range of network-based services in the future. We have demonstrated it in conjunction with multi-process servers as well as event-driven servers, and have shown that it can help defend these servers against a range of attacks. We are investigating its use for DNS servers, which tend to prefer UDP over TCP in order to reduce connection-related overheads, and for sensors on PlanetLab, which use an HTTP framework for simple information services. We expect that both environments will also prove amenable to Connection Conditioning.

Acknowledgments

We would like to thank our shepherd, David Andersen, and the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grant CNS-0519829.

References

196

- [1] ACME Laboratories. thttpd. http://www.acme.com/thttpd.
- [2] Akamai Technologies Inc. http://www.akamai.com/.
- [3] Apache Software Foundation. Apache HTTP Server Project. http://httpd.apache.org/.
- [4] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop* on *Internet Server Performance*, June 1998.
- [5] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.
- [6] G. Banga, J. C. Mogul, and P. Druschel. Resource containers: A new facility for resource management in server systems. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), February 1999.
- [7] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, 1999.
- [8] A. Chankhunthod, P. B. Danzig, C. Neerdaeles, M. F. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX Annual Technical Conference*, January 1996.
- [9] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In Sixth International World Wide Web Conference, April 1997.
- [10] FreeBSD Project. Remote denial-of-service when using accept filters.
- http://www.securityfocus.com/advisories/4159.

 [11] Germán Goldszmidt and Guerney Hunt. NetDispatcher: A TCP Connection Router. Technical report, IBM Research, Hawthorne, New York, July 1997. RC 20853.
- [12] J. Hu, I. Pyarali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance

- over high-speed networks. In *Proceedings of the IEEE GLOBE-COM '97*, November 1997.
- [13] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.
- [14] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In ACM Symposium on Theory of Computing, 1997.
- [15] J. Lemon. Kqueue: A generic and scalable event notification facility. In FREENIX Track: USENIX Annual Technical Conference, Boston, MA, June 2001.
- [16] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC '97)*, San Diego, CA, 1997.
- [17] Netcraft Ltd. Web server survey archives. http://news.netcraft.com/archives/web_server_survey.html.
- [18] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.
- [19] N. Provos. libevent. http://www.monkey.org/ provos/libevent/.
- [20] N. Provos. Improving host security with system call policies. In Proceedings of the 12th USENIX Security Symposium, Washington, DC, 2003.
- [21] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA USA, December 2002.
- [22] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. Raw data and independent analysis. http://www.measurement-factory.com/results/.
- [23] Y. Ruan and V. Pai. Making the "Box" transparent: System call performance as a first-class result. In USENIX Annual Technical Conference, Boston, MA, June 2004.
- [24] J. Salz, A. C. Snoeren, and H. Balakrishnan. TESLA: A transparent, extensible session-layer architecture for end-to-end network services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems(USITS '03)*, Seattle, WA, 2003.
- [25] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. In 18th International Teletraffic Congress (ITC 2003), August 2003.
- [26] M. Slemko. Possible security issues with Apache in some configurations. http://www.cert.org/vendor_bulletins/VB-98.02.apache.
- [27] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, 1999.
- [28] Standard Performance Evaluation Corporation. SPEC Web Benchmarks. http://www.spec.org/web99/.
- [29] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In Proceedings of the 11th Symposium on Operating System Principles (SOSP-11), Austin, TX, 1987.
- [30] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings* of the 19th Symposium on Operating System Principles (SOSP-19), Lake George, New York, October 2003.
- [31] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In 4th USENIX Conference on Internet Technologies and Systems (USITS'03), March 2003.
- [32] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of* the 18th Symposium on Operating System Principles(SOSP-18), Chateau Lake Louise, Banff, Canada, Oct. 2001.

PCP: Efficient Endpoint Congestion Control

Thomas Anderson, Andrew Collins, Arvind Krishnamurthy and John Zahorjan University of Washington

Abstract

In this paper, we present the design, implementation, and evaluation of a novel endpoint congestion control system that achieves near-optimal performance in all likely circumstances. Our approach, called the Probe Control Protocol (PCP), emulates network-based control by using explicit short probes to test and temporarily acquire available bandwidth. Like TCP, PCP requires no network support beyond plain FIFO queues. Our initial experiments show that PCP, unlike TCP, achieves rapid startup, small queues, and low loss rates, and that the efficiency of our approach does not compromise eventual fairness and stability. Further, PCP is compatible with sharing links with legacy TCP hosts, making it feasible to deploy.

1 Introduction

The efficient and fair allocation of distributed resources is a longstanding problem in network research. Today, almost all operating systems use TCP congestion control [27] to manage remote network resources. TCP assumes no network support beyond that packets are dropped when the network is overloaded; it uses these packet losses as a signal to control its sending rate. Provided that all endpoints use a compatible algorithm, persistent congestion can be averted while still achieving good throughput and fair allocation of the shared resource.

However, it has long been understood that TCP is far from optimal in many circumstances. TCP managed networks perform poorly for moderate sized flows on idle links [12, 28], interactive applications [21], applications demanding minimally variable response times [13], high bandwidth-delay paths [32], and wireless networks [5]. In each case, the response in the cited papers has been to propose explicit network support. Clearly, network-based resource allocation can be designed to perform optimally. Thus, the research debate has largely focused on the appropriate knobs to place in the network, and specif-

ically, the tradeoff between simplicity and optimality in network support for resource management.

We take a radically different approach. Our goal is to demonstrate that cooperating endpoints, without any special support from the network, can achieve near optimal resource allocation in all likely conditions. Our motivation is partly intellectual – what are the algorithmic limits to endpoint resource allocation? Our motivation is also partly practical – it is much easier to deploy an endpoint solution than to modify every router in the Internet. Since TCP congestion control was first introduced in 1988, three substantial changes to its algorithm have been introduced and widely adopted [38]; by contrast, to date no router-based changes to congestion management have achieved equally widespread use.

Our approach is simple: we directly emulate networkbased control. Our algorithm, called the Probe Control Protocol (PCP), sends a short sequence of probe packets at a specific rate to detect whether the network can currently support the test rate, given its current traffic load. If so, the endpoint sends at that rate; if not, e.g., if other hosts are already using the bandwidth, the endpoint tries a new probe at a slower rate. A key element is that we use rate pacing (instead of TCP-like ack clocking) for all traffic; this allows probes to be short and precise. These short probes are "low impact" compared to the TCP approach of sending at the target rate for the full round trip time. Thus, endpoints can be aggressive with respect to testing for available bandwidth, without causing packet loss for existing flows. For example, a new arrival can use history to safely guess the currently available bandwidth. Since most links are idle most of the time, this often allow endpoints to jump (almost) immediately to full utilization of the link.

We have implemented PCP as a user-level process on Linux. Initial tests on the RON testbed show that PCP can outperform TCP by an average of a factor of two for 200KB transfers over the wide area, without having any measurable impact on competing

	Endpoint	Router Support
Try and Backoff	TCP [27], Vegas [10] RAP [45], FastTCP [31] Scalable TCP [34] HighSpeed TCP [19]	DecBit [44], ECN [18] RED [21], AQM [9]
Request and Set	PCP	ATM [4, 46], XCP [32] WFQ [14], RCP [15]

Table 1: Congestion Control Design Space

TCP traffic. We supplement these results with simulation experiments, where we show that PCP achieves our goals of near optimal response time, near zero packet loss rates, and small router queues, across a wide variety of operating environments; we further show that it has better fairness properties than TCP. A Linux kernel implementation is also currently in progress. Interested readers can obtain the source codes for the implementations and the simulation harness at http://www.cs.washington.edu/homes/arvind/pcp.

A practical limitation of our work is that, like TCP, we provide no means to counter misbehaving hosts; network based enforcement such as fair queueing [14] is still the only known means to do so. We show that PCP, unlike TCP, benefits from fair queueing, thus making it the first system to do well for both FIFO and fair-queued routers. Any future network is likely to have a mixture of FIFO and fair-queued routers, making an endpoint solution compatible with all a necessity for network evolution. In our view, TCP's poor performance over fair-queuing routers is a barrier to further deployment of router enforcement. By contrast, PCP can be seen as a stepping stone for more robust isolation mechanisms inside the network, thereby improving the overall predictability of network performance.

The rest of this paper presents our approach in more detail. Section 2 outlines the goals of our work, arguing that TCP is sub-optimal in many common network conditions found today. Section 3 describes the design of the PCP algorithm. We evaluate our approach in Section 4, discuss related work in Section 5, and summarize our results in Section 6.

2 Design Goals

Table 1 outlines the design space for congestion control mechanisms. We argue in this section that PCP explores a previously unstudied quadrant of the design space – endpoint emulation of optimal router-based control. If we were to start from scratch, the design goals for a congestion control algorithm would be clear:

 Minimum response time. The average time required for an application transfer to complete should be as

- small as possible. Since most transfers are relatively short, startup efficiency is particularly important [16].
- Negligible packet loss and low queue variability. Because sources in a distributed system cannot distinguish between the root causes of packet loss, whether due to media failure, destination unavailability, or congestion, it is particularly important to avoid adding to that uncertainty. Similarly, large queueing delays unnecessarily delay interactive response time and disrupt real-time traffic.
- Work conserving. In steady state, resources should not be left idle when they might be used to send data.
- Stability under extreme load. Aggregate performance should approach physical limits, and per-flow performance should degrade gracefully as load is added to the system.
- Fairness. Competing connections (or flows) which are not otherwise limited should receive equal shares of the bottleneck bandwidth. At the very least, no flow should starve due to competing flows.

Note that network-based congestion control can easily achieve all of these goals [4, 46]. With ATM, for example, endpoints send a special rate control message into the network to request bandwidth, enabling the bottleneck switch or router to explicitly allocate its scarce capacity among the competing demands. We call this approach "request and set" because endpoints never send faster than the network has indicated. Response time is minimized because fair share is communicated in the minimum possible time, a round trip. This also results in queues being kept empty and bandwidth being allocated fairly. Our goal is to see if we can achieve all these properties without any special support from the network [47], by emulating "request and set" mechanics from endpoints.

By contrast, TCP congestion control achieves the last three goals [27] but not always the first two. TCP carefully coordinates how the sending rate is adjusted upwards and downwards in response to successful transmissions and congestion signals. We call this approach "try and backoff" since an end host sends traffic into the network without any evidence that the network has the capacity to accept it; only when there is a problem, that is, a packet loss, does the end host reduce its rate.

Since a TCP endpoint has no knowledge of the true available bandwidth, it initially starts small and through a series of steps called slow start, drives the network to saturation and packet loss, signaling the capacity limit of the network. Although effective, this process can waste bandwidth on startup – asymptotically $O(n \log n)$ in terms of the path's bandwidth delay product [12]. (To be fair, TCP congestion control was designed at a time

when links were thin and usually fully utilized; in these situations the efficiency loss of slow start is minimal.) Further, TCP's slow start inefficiency is fundamental. Several proposals have been made for methods to jump start the initial window size, but they run the risk of causing increased packet losses in situations where there is persistent congestion [19].

Once a TCP endpoint determines the available bandwidth, in theory the link will be fully utilized, amortizing the initial inefficiency for a sufficiently long connection. Of course, many flows are short. Even for long flows, TCP steady state behavior can be disrupted by the bursty traffic pattern emitted by other flows entering and exiting slow start. In practice, TCP may achieve only a fraction of the available bandwidth, because of the need to slowly increase its sending rate after a loss event to avoid persistent congestion [30, 32]. Similar problems occur with TCP in the presence of noise-induced loss, such as with wireless links [5].

Some researchers have studied how to modify end hosts to improve TCP performance, while keeping its basic approach. For example, TCP Vegas [10], FastTCP [31], Scalable TCP [34], and HighSpeed TCP [19] all attempt to improve TCP steady state dynamics. Vegas and FastTCP are similar to PCP in that they use packet delay to guide congestion response, but unlike PCP, they do so only after sending at the target rate for the entire round trip time. And because all of these alternate approaches leave slow start unchanged, they only help the small fraction of transfers that reach steady state.

Other researchers have explored adding TCP-specific support to routers. For example, routers can drop packets before it becomes absolutely necessary [21, 9], as a way of signalling to end hosts to reduce their sending rates. However, this trades increased packet losses in some cases for lower delay in others; most users want both low loss and low delay. Some have advocated setting a bit in a packet header to signal congestion back to the sender [44, 18], but this does nothing to address the slow start inefficiency. This has led some to advocate adding an ATM-style rate control message to the Internet to allow for more rapid TCP startup [42, 15]. And so forth.

Our approach is to explore the opposite quadrant in Table 1. Is it possible to achieve all five goals using only endpoint control, by emulating the "request and set" semantics of explicit router-based resource allocation? In doing so, we hope to design a system that is better suited to the tradeoffs we face today vs. what TCP faced fifteen years ago. In designing our system, note that we place the first two goals listed above ahead of the last three, in priority. While we want our system to to be efficient, stable and fair under high load, we also want our system to

behave well in the common case.

The common case is that most network paths are idle most of the time, and are becoming more so over time [41, 2]. This was not always true! Rather, it is the natural consequence of the cumulative exponential improvement in the cost-performance of network links—at least in industrialized countries, it no longer makes sense for humans to wait for networks. By contrast, when TCP congestion control was initially designed, wide area network bandwidth cost over one thousand times more than it does today; at that price, fairness would naturally be more important than improving connection transfer time. The opposite holds today.

Second, even with HTTP persistent connections, most Internet transfers never reach TCP steady state [22, 48]. Even when they do, startup effects often dominate performance [12]. For example, a 1MB cross-country transfer over fast Ethernet can achieve an effective throughput with TCP of only a few Mbps, even with no other flows sharing the path. Home users are more frequently bandwidth-limited, but even here, TCP is not well-suited to a highly predictable environment with little multiplexing.

Third, computation and memory are becoming cheaper even faster than wide area network bandwidth. TCP was originally designed to avoid putting a multiplication operation in the packet handler [27], yet at current wide area bandwidth prices, it costs (in dollars) the same amount to send a TCP ack packet as to execute half a million CPU instructions [25]. One consequence is that hardware at aggregation points is increasingly limited by TCP mechanics: to remain TCP "friendly" the aggregation point must not send any faster than k parallel TCP connections [6], something that is only efficient if there are multiple active flows to the same destination. By contrast, PCP can benefit directly from an endpoint's excess cycles and memory by modeling the likely behavior of a network path, even if the path has not been recently used.

Finally, our approach integrates better with constant rate real-time traffic. Without router support, TCP's continual attempts to overdrive and backoff the bottleneck link can disrupt fixed-rate flows that are sharing the link, by introducing packet delay jitter and loss. To some extent, this problem can be reduced by sophisticated active queue management (AQM) [24]. Lacking widespread deployment of AQM systems, most ISP's today have abandoned the vision of integrated services—they provision logically separate networks to carry voice over IP as distinct from regular web traffic—as the only practical way to achieve quality of service goals. By contrast, in PCP, best effort traffic will normally have very little impact on background fixed-rate traffic, again without any special hardware support.

Mechanism	Description	Goal	Section
probes	Senders use short transmission bursts with limited payload to "prove" the existence of available bandwidth, while minimizing any long term effects of failed tests.	low loss	Section 3.1
direct jump	Given a successful test, senders increase their base rate to the rate that test.	min response time	Section 3.1
probabilistic accept	Accept tests taking into account the variance observed in the available bandwidth measurements.	fairness	Section 3.1
rate compensation	When existing senders detect increasing queueing, they reduce their rates to drain the queue.	low loss, low queues	Section 3.2
periodic probes	Senders periodically issue new probes to try to acquire additional bandwidth.	work-conserving	Section 3.3
binary search	Senders use binary search to allocate the available bandwidth.	min response time, work conserving	Section 3.3
exponential backoff	Senders adjust the frequency of tests to avoid test collisions and failures.	stability	Section 3.3
history	Senders use heuristics to choose the initial probe rate.	min response time	Section 3.4
tit for tat	Reduce speed of rate compensation if past compensation was ineffective.	TCP compatibility	Section 3.5

Table 2: A Summary of PCP Mechanisms

To sum, TCP is optimized for the wrong case. Like TCP, the design we describe in this paper provides robust congestion control for a wide range of operating conditions. But equally importantly, our approach is better than TCP for the common case: moderate-sized transfers over mostly idle links with near-zero loss and low delay even when there is congestion. Of course, network-based congestion control solutions have already been shown to provide these characteristics; our point is simply to demonstrate that we can achieve these goals without network support.

3 PCP Design

In this section, we sketch the various elements of the PCP design. Table 2 provides a road map. For this discussion, we assume that PCP is used by all endpoints in the system to manage network resources; we defer to the end of this section a discussion of how to make PCP backwardly compatible with TCP end hosts. PCP represents a clean-slate redesign to endpoint congestion control; we do however retain the TCP mechanisms for connection management and flow control.

3.1 Emulating Request-and-Set

PCP is very simple at its most basic (Figure 1): endpoints send *probe packets*, which are short sequences of packets spaced at a target test rate, to determine if the network has the capacity to accommodate the request. If this *probe* is successful, the end host can immediately increase its base rate by the target rate of the probe; it

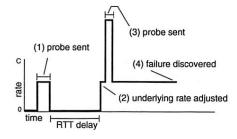


Figure 1: Example of a successful and a failed PCP probe.

then transmits the *baseline packets* in a *paced* (equally spaced) manner at the new *base rate*. The probe rate is adjusted (primarily) by changing the duration over which the probe data is sent, not the amount of data that is sent. If the initial probe is unsuccessful (e.g., the network does not have the spare capacity), the end host must try again. A key element of our approach is that endpoints only increase their base sending rates immediately after a successful probe, and at no other time; thus, modulo the round trip delay, a successful probe indicates that the network resource is unallocated.

Probe success and failure is defined by whether the probe packets induce queueing inside the network, measured by whether the delay increases during the probe. For each PCP data packet, a timestamp is recorded at the receiver and returned to the sender, akin to the TCP timestamp option [26]; measuring changes in delay at the receiver allows us to eliminate variability induced

by the reverse path. Our test for available bandwidth is similar to the one suggested by Jain and Dovrolis [29], but is designed to be more robust to the noise caused by PCP's probe process. Specifically, we use least squares to fit a line through the sequence of delay measurements and accept the test rate if the measurements are consistent with flat or decreasing delay; otherwise, the probe is rejected. Furthermore, since measurements are rarely determinative of the true state of the network, we use a "probabilistic accept" rule that accepts probes nondeterministically. The least squares fit yields a probability distribution function characterized by the estimated delay slope and a standard error for the estimated value. A low error indicates a good fit, while a high value might be due to measurement noise or variability in cross traffic. We randomly choose whether to accept a probe based on this probability distribution.

We do not assume a hard real-time operating system implementation; some jitter is acceptable in the scheduling of packets as the fitting process is robust to small variations in packet spacing. We however assume the availability of fine-grained clocks; nanosecond clocks and timers have become commonplace on modern processors [40], sufficient for scaling to gigabit speeds. Generic timestamp and packet pacing logic is also becoming increasingly common on network interface hardware.

To enable an apples to apples comparison, we set the initial probe size to match the initial TCP packet. After an initial packet exchange to verify the receiver is willing to accept packets (a mirror of the TCP SYN exchange), PCP sends, as its first probe, data equal to a maximum size packet, but divided into k separate chunks (currently, 5). Probe packets may carry live data, so that if the data to be transferred is sufficiently small, it may complete within the probe packets - that is, whether or not the network can accept packets indefinitely at the probe rate. Although some have proposed using a larger initial window size in TCP to speed its discovery of the available bandwidth for high capacity paths, this would come at the potential cost of added congestion on low capacity paths. By separating the probe rate from the size of the probe, PCP avoids having to make this trade-off; as we describe below, we can safely use history to select an aggressive rate for the initial test.

If the probe is successful, PCP immediately jumps to the requested rate. As a minor optimization, once an end host is successful in obtaining sufficient bandwidth, we convert to using k maximum sized packets as probes, again, paced at the target bit rate. This provides better accuracy for high bandwidth paths.

3.2 Rate compensation

A key challenge for PCP is to gracefully eliminate queues that might build up at a bottleneck router. Queues can build up for several reasons. One obvious cause is failed probes. If all of the bottleneck bandwidth has been allocated, any additional probe will induce queueing that will not disappear until some flow reduces its bandwidth. As more failed probes accumulate, the queues could slowly build to the point where packet loss is inevitable. A more severe cause is due to the time lag between when an end host makes a probe and when it can allocate the bandwidth. In PCP, the request and set operations are not atomic. If two or more hosts send a probe at approximately the same time, both probes may succeed, resulting in duplicate allocation of the same bandwidth. In this case, the link may be over committed, and unless one or both hosts reduce their rates, queues will quickly build up to cause packet loss.

Fortunately, it is straightforward for PCP to detect queueing at the bottleneck. Recall that once an end host allocates bandwidth, it sends its data as paced packets at the base rate. If there is no queueing at the bottleneck, the delays for these data packets will be regular. Any increase in the delay indicates a queue, requiring a rate reduction to eliminate. Note that complex link layer arbitration, as in 802.11, is perfectly compatible with PCP; any added delay in those systems is an indication of queueing – that the endpoints are sending faster than the underlying network can handle.

Eliminating queues caused by PCP endpoints is also easy. Whenever a queue is detected, all existing senders proportionately reduce their rate sufficiently to eliminate the queue over the next round trip. We call this process, *rate compensation*. Eliminating the queue over one round trip is more aggressive than is strictly required by control theory [32], but in our system, any queueing is an indication of resource contention. Under contention, proportionate decrease among existing senders, and uniform competition for newly available bandwidth among all senders, helps achieve eventual fairness.

We use two mechanisms to detect over-committed network links. First, we monitor the gap between baseline PCP packets as they enter and exit a network path. If the time gap observed at the receiver (Δ_{out}) is greater than the spacing Δ_{in} used by the sender, the bottleneck link is likely to be overloaded. To avoid making matters worse, we reduce the base sending rate by a factor of $(\Delta_{out} - \Delta_{in})/\Delta_{out}$. Second, in order to drain the queue, we monitor the one-way delays experienced by PCP packets. If the maximum one-way delay (maxdelay) observed in the previous round trip time is greater than the minimum observed one-way delay to the destination (min-delay), then there is persistent queueing at the bottleneck link. To eliminate the queue build-up, we

reduce the sending rate by a factor of (max-delay - min-delay)/max-delay. In both cases, we bound the proportionate decrease to the TCP backoff rate – no more than half of the prior rate during any round trip. (We concern ourselves only with the measurements during the previous round trip as prior rate compensation is presumed to have eliminated the overloads during prior round trips.) If all senders detect queueing and reduce their rate by this proportion, and no further probes are launched, it is easy to show that the queue will disappear within one round trip time. Senders with much shorter round trip times will reduce their rate more quickly, shouldering more of the burden of keeping queues small, but they will also be able to acquire bandwidth more quickly by probing for new bandwidth at a faster rate.

Once the base rate is reduced, probes may successfully re-acquire the bandwidth. These probes may be launched either by other nodes, or even by the reducing node itself. This is done to foster additive increase, multiplicative decrease behavior when there is contention. If the queues do not dissipate, the existing senders will continue to proportionally decrease their rates. Some combination of flows will acquire the released bandwidth.

A detail is that we apply the rate compensation incrementally, after every acknowledged packet, by comparing the required rate compensation to the rate reductions that have already been applied over the previous round trip time. If the new rate compensation is larger, we reduce the sending rate by the difference. This is similar to a single rate adjustment made once per round trip time, but operates at a finer granularity. Further, to reduce the impact of the noise on the system, we discard outliers represented by either the lowest 10% or the highest 10% of the measured values.

Another detail is that Internet routing changes can transparently alter the baseline one-way packet delay. Although some have called for providing endpoints the ability to detect when their packets have been rerouted [36], that facility is not available on the Internet today. There are two cases. If the routing change decreases the baseline delay, the node will update its mindelay, observe there is a difference between the new mindelay and the previous max-delay, and proceed to reduce its rate by at most one half. Behavior will then revert to normal after one round trip, and the end host will be free to probe to acquire bandwidth on the new path. If the routing change increases the baseline delay, the node will see an increase in its max-delay and likewise reduce its rate in an attempt to compensate. This reduction will dissipate after min-delay has timed out. Note that the probe process is independent of rate compensation; probe success is based on the measured increase in delay during the probe, and not on the long term estimation of the queue. Thus, as long as the new path has spare capacity, the end host will be able to quickly re-acquire its released bandwidth. Both types of routing changes result in temporarily reduced efficiency, but with the positive side effect that the affected endpoints are less aggressive exactly when the state of the network is in flux.

3.3 Probe control

Because probes carry limited payload and rate compensation corrects for any mistakes that occur, an end host can be aggressive in selecting its probe rates. For example, it can pick the probe rate that maximizes its expected yield - the likelihood of the probe's success times the requested rate. Unless there are long periods of full utilization of the network, it is better for all concerned for an arriving flow to quickly grab all available bandwidth, complete the transfer, and exit the system, leaving future resources for future arrivals. Of course, we do want the system to be work-conserving, stable and fair under persistent congestion, and thus we need to introduce additional mechanism in PCP to accomplish those goals. That is the topic of this sub-section. Note however that if high load behavior is your only concern (e.g., response time and loss rate are unimportant), TCP's current behavior is adequate, and our design would offer few benefits.

In the absence of any other information, we set the initial target probe rate to be one maximum sized packet in half of the round trip time, as measured by the initial connection establishment (TCP SYN) packet exchange. If successful, during the next round trip the end host can send its base rate packets at that probe rate – two maximum sized packets per round trip time. It may also continue probing.

In our initial prototype, we use exponential increase and decrease to guide the search process, doubling the attempted rate increase after each successful probe, and halving the rate increase after each unsuccessful one. This guarantees that, regardless of the starting point or the capacity of the link, an end host fully allocates the available bandwidth in $O(\log n)$ steps. (A further optimization, which we have not vet implemented, is to use the slope of the response times from a failed probe to guess the next probe rate - in essence, the slope tells us by how much we have overestimated the available bandwidth. This will be particularly important for capacityconstrained paths, as the initial round trip time as measured by the small connection establishment packet, may vastly underestimate the round trip time for a maximally sized packet.) Note that while we never conduct more than one probe per measured round trip time, if the available bandwidth is small enough, we may stretch a single sequence of probe packets across multiple round trips. Thus, PCP gracefully scales down to very low bandwidth and/or oversubscribed paths. By contrast, TCP has a minimum window size of one packet; this can result in very high packet loss rates for paths where each flow's share is less than a single packet per round trip [39, 37].

Since we do not want nodes to continuously probe unsuccessfully for bandwidth, we place a lower bound on the rate that a flow can request from the network, at 1% of its current base rate. Once an existing sender has failed to acquire its minimum rate, it exponentially reduces its frequency of probing, up to a limit of 100 round trips. Within this interval, the probe is placed randomly. This is analogous to Ethernet, for a similar purpose. Removing the limit would improve theoretical scalability, but at a cost of reduced efficiency in acquiring recently released bandwidth.

We further note that our probabilistic accept rule for probes allows a probe to succeed even if it causes a small amount of queueing, thereby allowing new connections to succeed at receiving small amounts of bandwidth and triggering incumbent flows to release their bandwidth due to rate compensation. As a side effect, the rule also improves efficiency under heavy load by keeping the queue non-empty [10]. This behaves much like router-based active queue management (AQM), but controlled from the endpoint. A side effect, discussed below, is that the rule also makes PCP more robust when competing with legacy TCP flows.

3.4 History Information

As described so far, PCP seems merely to have replicated the delays caused by TCP slow start $-O(\log n)$ steps to determine the bandwidth. However, we note that the impact of a PCP probe is independent of its test rate, in contrast to TCP's approach of sending at its target rate for a full round trip time. Thus, it is easy to test aggressively in PCP, without fear of disrupting existing connections. We use this flexibility to reduce the startup transient to a constant number of round trips in the common case.

We achieve this by keeping history information about the base rates previously used to each Internet address. When this history information is available, we set the initial probe rate to be 1/3 of the previous base rate, and then use binary search from that point forward. This allows the end host to usually identify the optimal rate within two round trip times after the initial connection establishment. We also keep track of the variance in base rates and the accuracy of predictions made based on the history information; if the history provides inaccurate estimate, we halve/double the initial probe rate after each inaccurate/accurate prediction, up to 1/3 of the base rate. Note that we do not set the initial rate to be the entire previous base rate, to avoid the chance that multiple hosts will simultaneously probe for, and seemingly acquire, the full link bandwidth. Our use of history is similar to that of the MIT Congestion Manager (CM) [6] but more general; because CM uses TCP, it can only reuse the congestion window if multiple transfers to the same location happen almost simultaneously. A mistakenly large congestion window in TCP could cause massive packet loss. Because the cost of making a mistake with history in PCP is only a wasted probe, we can make more aggressive use of potentially stale data of the likely available bandwidth of a path.

3.5 TCP Compatibility

To be feasible to deploy, PCP must be able to share network resources with existing TCP connections. Naively, as TCP increases its window size, queues will build up, and any PCP endpoints will reduce their sending rate to compensate. The TCP host will proceed unimpeded, continuing to increase its window size, causing further rate reductions from the PCP hosts.

We do not believe it is essential for PCP to be strictly fair with respect to TCP hosts, since that might well require simulating TCP's precise semantics. Rather, our goal is for PCP to be incentive compatible when sharing resources with TCP, so that it outperforms TCP while avoiding starvation for TCP hosts. Since PCP does much better than TCP for the common case of short to moderate transfers, there is substantial room for PCP to outperform TCP without needing to actively penalize TCP senders.

Our design walks this delicate balancing act. First, we recognize when a bottleneck is being shared with TCP, by observing when PCP's rate compensation is ineffective at reducing the queue size over several consecutive round trips. Normally PCP would continue to decrease its rate in the hope of eliminating the queue, but instead we apply a "tit for tat" rule, decreasing the rate compensation by a factor of ten for as long as rate compensation is ineffective. While this might seem counter-intuitive - increasing aggressiveness precisely at the point when congestion is building - "tit for tat" is needed to counter TCP's overly aggressive behavior. Eventually, the TCP connection will over-drive the link, causing loss and backoff for the TCP sender; PCP can then increase its rate during these periods. Our measurements and simulation results indicate that the TCP backoff and reduced PCP rate compensation balance out in most cases. When the TCP flow completes, any remaining PCP flows will find that rate compensation again becomes effective, enabling them to revert to their normal behavior.

In all other respects, PCP is backwardly compatible with legacy TCP hosts. We re-use the TCP packet header, indicating whether PCP should be used as an option in the TCP SYN packet. If the PCP receiver acknowledges the option, the sender uses PCP; otherwise we use traditional TCP congestion control.

There is no fundamental reason we cannot design a PCP sender to interoperate with an unmodified TCP re-

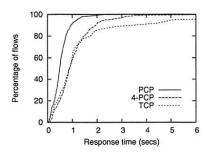


Figure 2: Cumulative distribution function of the average transfer time for the 380 wide-area paths in the RON testbed.

ceiver. The principal difference between a PCP and a TCP receiver are in the precise semantics of timestamps and delayed acknowledgments. TCP timestamps are similar in theory to those used in PCP to measure one-way delay, but the TCP specification is tightly bound to the TCP retransmit timer logic. The TCP timestamp reflects the time that the *data* being acknowledged was received, not the time that the packet causing the acknowledgment was received. Instead, we plan to use round trip measurements to approximate one-way delay when interoperating with a TCP receiver. Similarly, PCP assumes that delayed acknowledgments are turned off; a PCP sender can disable the receiver's delayed acknowledgment logic by simply reordering every other packet or by sending all probe packets as doublets.

An interesting, and future, research question is whether we can design a PCP receiver to induce a TCP sender to use PCP congestion control. Savage et al. [49] have shown that a malicious receiver can abuse a sender's TCP control logic to cause it to send at an arbitrary rate; we believe we can leverage those ideas for inducing PCP compatibility with legacy TCP senders.

4 Evaluation

In this section, we first present data from a user-level implementation of PCP; we then use simulation to examine the behavior of PCP in more detail. Our results are both preliminary and incomplete; for example, we provide no study of the sensitivity of our results to the choice of PCP's internal parameters.

4.1 Performance Results from a User Level Implementation

This section presents data for PCP and TCP transfers over the Internet between twenty North American nodes selected from the RON testbed [3].

We implemented the PCP protocol in a user-level process; this is a conservative measure of PCP's effectiveness, since timestamps and packet pacing are less accurate when done outside the operating system kernel. To enable an apples-to-apples comparison, we also imple-

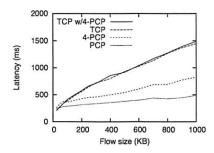


Figure 3: Transfer times for the user-level implementation.

mented TCP-SACK in the same user-level process. With TCP-SACK, selective acknowledgments give the sender a complete picture of which segments have been received without loss. The sender uses fast retransmit whenever it receives three out-of-order acknowledgments and then enters fast recovery. Our TCP implementation assumes delayed acknowledgments, where every other packet arriving within a short time interval of 200 ms is acknowledged. As in most other implementations [38], acknowledgments are sent out immediately for the very first data packet (in order to avoid the initial delayed ACK timeout when the congestion window is simply one) and for all packets that are received out-of-order. Removing delayed ACKs would improve TCP response time, but potentially at a cost of worse packet loss rates by making TCP's slow start phase more aggressive and overshooting the available network resources to a greater extent. We used RON to validate that our user-level implementation of TCP yielded similar results to native kernel TCP transfers for our measured paths.

For each pair of the twenty RON nodes, and in each direction, we ran three experiments: a single PCP transfer, a single TCP-SACK transfer, and four parallel PCP transfers. Each transfer was 250KB, repeated one hundred times and averaged. Figure 2 presents the cumulative distribution function of the transfer times for these 380 paths. PCP outperforms TCP in the common case because of its better startup behavior. The average PCP response time is 0.52 seconds; the average TCP response time is 1.33 seconds. To put this in perspective, four parallel PCP 250KB transfers complete on average in roughly the same time as a single 250KB TCP transfer. Further, worst case performance is much worse for TCP; while all PCP transfers complete within 2 seconds, over 10% of TCP transfers take longer than 5 seconds. While this could be explained by PCP being too aggressive relative to competing TCP traffic, we will see in the next graph that this is not the case. Rather, for congested links, TCP's steady state behavior is easily disrupted by background packet losses induced by shorter flows entering and exiting slow start.

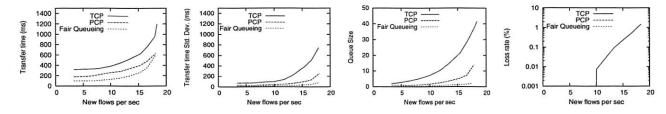


Figure 4: Performance of TCP, PCP and fair queueing. File transfer time includes connection setup. Bottleneck bandwidth is 40 Mb/s, average RTT is 25 ms, fix ed length flows of 250 KB. Fair queueing and PCP suffer no packet loss; the corresponding lines in the loss rate graph overlap with the x-axis.

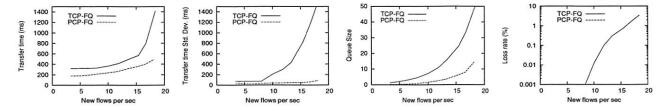


Figure 5: Performance of TCP and PCP through a bottleneck router that implements fair queueing. Bottleneck bandwidth is 40 Mb/s, average RTT is 25 ms, fix ed length flows of 250 KB. Fair queueing and PCP suffer no packet loss.

In Figure 3, we examine the behavior of PCP and TCP as a function of flow size, for a single pair of nodes. The TCP behavior is dominated by slow start for small transfers, and steady state behavior for large transfers; since this path has a significant background loss rate caused by other TCP flows, TCP is unable to achieve more than 5Mb/s in steady state. By contrast, PCP is able to transfer large files at over 15Mb/s, without increasing the background loss rate seen by TCP flows. To show this, we ran four parallel PCP transfers simultaneously with the TCP transfer; the TCP transfer was unaffected by the PCP traffic. In other words, for this path, there is a significant background loss rate, limiting TCP performance, despite the fact that the congested link has room for substantial additional bandwidth.

While some researchers have suggested specific modifications to TCP's additive increase rule to improve its performance for high bandwidth paths, we believe these changes would have made little difference for our tests, as most of our paths have moderate bandwidth and most of our tests use moderate transfer sizes. Quantitatively evaluating these alternatives against PCP is future work.

4.2 Simulation Results

We next use simulation to examine PCP's behavior in more detail. We chose not to use ns-2 for our simulations as it tends to be slow and scales poorly with the number of nodes or flows in the system. Using our own simulator also enabled us to reuse the same code for PCP and TCP that we ran on RON. Recall that we validated our TCP implementation against the TCP in the RON kernel.

For comparison, we also implemented centralized fair

queueing [14] in our simulator. This mechanism achieves near-perfect isolation and fairness by scheduling packets from active flows in a bit-sliced round robin fashion. Given such a centralized router mechanism, it is possible for endpoints to infer their fair share by sending a pair of back to back packets, and observing their separation upon reception [35]. We model a fair-queueing system where the router also assists the endpoint in determining the optimal sending rate in the following manner. The endpoint transmits a control packet every RTT, and the router tags this packet with the flow's fair bandwidth allocation and the current queue for the flow. The endpoint simply sends at its fair share rate while compensating for queue buildups resulting from dynamic changes in the number of flows. While the resulting design might be difficult to realize in practice as it requires the router to communicate multiple bits of congestion information to the endpoint, it is intended as a bound on what is possible.

4.2.1 Impact of Varying Offered Load

We begin by evaluating the effect of varying load on the performance of our proposed algorithm and its alternatives. We consider a simple topology with a single bottleneck shared by many competing flows. The bottleneck bandwidth is 40 Mb/s and is shared by a hundred source-destination pairs. We model the bottleneck router as a FIFO drop-tail router for TCP and PCP flows. The buffering at the bottleneck is set to the bandwidth delay product. The round trip times for the source-destination pairs are uniformly distributed from 15 ms to 35 ms. We simulate fixed-length flows of 200 packets of size 1250

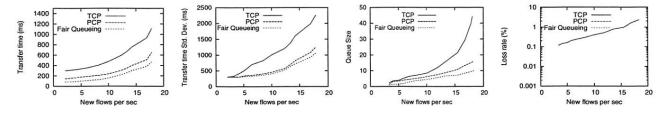


Figure 6: Performance when flow lengths and inter-arrival times are Pareto distributed. Mean flow length is 250 KB, bottleneck bandwidth is 40 Mb/s, and average RTT is 25 ms. Note that the standard deviation plot is depicted on a different scale due to the higher variances caused by Pareto-distributed flow lengths. Fair queueing and PCP suffer no packet loss.

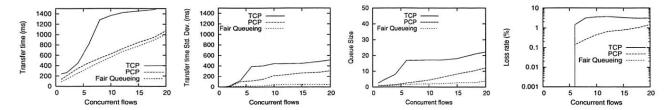


Figure 7: Performance with synchronized start of flows. Flow length is 250 KB, bottleneck bandwidth is 40 Mb/s, and average RTT is 25 ms. Fair queueing suffers no packet loss.

bytes each (resulting in an overall flow size of 250 KB). We vary the arrival rate of new flows to control the offered load to the system. The flow arrivals are randomized, using a Poisson arrival process, to avoid synchronization artifacts and to simulate varying load conditions, but the mean arrival rate is fixed based on the desired level of load.

The simulation parameters for this and the following experiments were chosen to be illustrative; they are not intended to be representative. For instance, instead of using a mixture of flow sizes as with real workloads, we typically use fixed size flows so that we can compare mean response times.

Figure 4 presents the variation in key performance characteristics as we increase the offered load from 15% to about 90% of bottleneck capacity. Since each flow offers 250 KB or 2 Mb of load, three new flows per second implies an offered load of 6 Mb/s or 15% of bottleneck capacity. We measure the response time to complete each transfer, including the cost of connection establishment. We also report the variability in response time (fairness), the queue size at the bottleneck, and the average loss rate at the bottleneck.

The results show that PCP has better response time than TCP and exhibits smaller variations in response time. PCP's response time is close to that of fair queueing. For low load conditions, PCP adds about two round-trip delays to fair queueing as it probes for available bandwidth before ramping up to the sending rate. At high load, PCP continues to perform well, ensuring that the bottleneck is kept busy; PCP keeps queues small while

holding down average response time. Across the spectrum of load conditions, PCP keeps queue sizes lower than TCP, and has no packet loss even at high loads. Even at moderate loads, packet losses can dramatically penalize specific TCP flows [12]; PCP avoids this effect.

4.2.2 TCP and PCP over Fair Queueing Routers

We next study the performance of TCP and PCP flows when they are transmitted through routers that implement fair queueing. Our goal is to show that PCP is compatible with and benefits from general-purpose network enforcement. Others have argued for TCP-specific rules for penalizing misbehaving endpoints [20], but we argue that this unnecessarily constrains the choice of end host algorithm.

In theory, fair queueing is neutral to the choice of endpoint congestion control algorithm. An endpoint that sends faster than its fair share will build up queues and cause packet loss, but only to its own packets. However, the inefficiency imposed by TCP slow start has no benefit if the bottleneck resource is fair queued, but the endpoint has no way in general of knowing how the network is being managed.

We use the same simulation parameters as those used in the previous experiment, but substitute the FIFO droptail router with a fair-queued router. Figure 5 shows that PCP benefits from fair queueing, with lower response time variance even at high loads. Bandwidth probes initiated by arriving PCP flows are allowed to succeed due to the isolation of flows by the fair-queued router. TCP, on the other hand, performs slightly worse with fair queueing. When a TCP flow exceeds its fair share of router

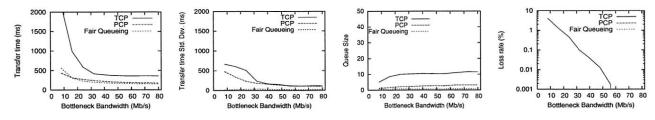


Figure 8: Effect of varying bottleneck bandwidth. Average RTT is 25 ms, flow lengths are 250 KB, and interarrival times are set to operate the system at 60% load. Fair queueing and PCP suffer no packet loss.

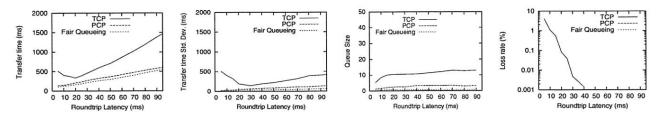


Figure 9: Effect of varying round-trip time. Bottleneck bandwidth is 40 Mb/s, flow lengths are 250 KB, and interarrival times are set to operate the system at 60% load. Fair queueing and PCP suffer no packet loss.

buffer space under loaded conditions, it suffers multiple losses, causing its performance to drop severely; in contrast, a drop-tail router spreads the packet losses more smoothly across all of its flows.

4.2.3 Bursty Traffic Patterns

We now evaluate the performance of PCP under bursty settings, where the history information might potentially yield incorrect predictions. First, we repeat the experiment from the previous section using a mixture of flow lengths instead of fixed length flows. We retain the mean length of flow to be 250 KB, but vary flow lengths according to a Pareto distribution with shape 1.25. We also set inter-arrival times to follow a Pareto distribution with shape 1.2. These parameters model self-similar traffic observed in real traces. Figure 6 plots the results. PCP provides performance close to router-based control in this case.

We next evaluate the different protocols when a number of concurrent flows are repeatedly started together resulting in synchronized transfers. In addition, we artificially rig PCP's history mechanism to always provide the estimate that all of the bottleneck bandwidth is available to each flow. By initiating multiple flows simultaneously with this estimate, we intend to evaluate PCP's performance when the history mechanism provides grossly inaccurate estimates. Figure 7 depicts the results as we vary the number of synchronized flows, each transferring 250 KB through our base configuration comprising of a 40 Mb/s bottleneck and 25 ms average RTT. Since the flow start times are synchronized, the TCP flows enter slow-start phase together. When the number of concurrent TCP flows is six or more, their combined ramp-up

during slow-start results in filling up the router queue and causes a large number of packet losses. PCP also suffers from packet loss due to inaccurate history information, most of which occurs during the first RTT after transitioning to a high sending rate. But the sending rate is throttled in the subsequent round-trip as the PCP sender performs rate compensation in response to increased inter-packet gaps and one-way delays.

4.2.4 Impact of Varying Simulation Parameters

We then perform sensitivity analysis to study the effect of varying the parameters of the topology. Figure 8 presents the various metrics as the bottleneck bandwidth is varied. The rate of flow arrivals is set such that the offered load to the system is 60% of the bottleneck capacity for the various runs of the experiment. At lower capacities, TCP's slow-start phase overruns the available bandwidth resources, causing packet loss storms, resulting in substantial back-off and increased transfer times. TCP's transfer time performance levels out with increasing bandwidth, but never approaches the performance of PCP due to the $O(\log n)$ overhead associated with the startup phase.

Figure 9 illustrates the performance of various flows through our base configuration of a 40 Mb/s bottleneck router as we vary the round-trip latency of the flows. We again consider fixed-size flows of length 250 KB, and we also fix the offered load at 60% (twelve new flows per second for this configuration). The average round-trip latency is varied from 5ms to 100ms, and the buffer space is set to the corresponding bandwidth-delay product for each run. At small RTTs, TCP flows tend to blow out the small router queues rather quickly, while at

high RTTs, the $O(\log n)$ slow-start overhead translates to much higher transfer times. PCP flows track the performance of fair queueing under all RTT conditions.

We also study performance as we vary the mean flow length. Figure 10 graphs the various performance metrics as we vary the flow size and correspondingly vary the arrival rate in order to fix the offered load at 60%. As we study the performance of TCP flows, we observe a tradeoff between two competing phenomena. As we increase the flow lengths, the initial slow-start overhead is amortized over a larger transfer. The resulting efficiency is however annulled by increased loss rates as there are a sufficient number of packets per flow for TCP to overrun buffer resources during the slow-start phase.

4.2.5 Impact of Transmission Loss

Finally, we evaluate the impact of transmission loss on the response time for the different protocols. We consider transmissions with an average RTT of 25 ms and subject the packets to a constant loss rate, independent of the load in the system. Figure 11 graphs the results. The response time for TCP blows up with increased loss rates, since TCP interprets losses as signals of congestion. PCP and fair queueing can tolerate losses without suffering a substantial increase in response times. In PCP, when a loss is detected, either through a timeout or by the presence of acknowledgments for subsequent messages, the packet is scheduled for retransmission for the next available time slot based on the current paced transmission rate.

5 Related Work

As we have noted, many of the elements of PCP have been proposed elsewhere; our principal contribution is to assemble these ideas into a system that can emulate the efficiency of network-based congestion control.

Our work on PCP is inspired in many ways by Ethernet arbitration [8]. PCP, like Ethernet, is designed to perform well in the common case of low load, with high load stability and fairness an important but secondary concern. Ethernet's lack of stability and fairness in certain extreme cases yielded much followup work within the academic community, but Ethernet's common case performance and simplicity was sufficient for it to be wildly successful in practice.

Our work also closely parallels the effort to define algorithms for endpoint admission control [11, 23, 33, 17, 7]. In these systems, endpoints probe the network to determine if a fixed-rate real-time connection can be admitted into the system with reasonable QoS guarantees and without disrupting previously admitted connections. As with our work, this research demonstrated that endpoints can effectively emulate centralized resource management. Nevertheless, there are significant differences with

our work. First, real-time connections have fixed bandwidth demands and are relatively long-running; hence, probes were run only at connection setup, and it was not necessary to make them particularly efficient. For example, Breslau et al. suggest that probes should use TCP-like slow start to determine if there is sufficient capacity for the connection [11]. Our system is designed to allow probes to be short and precise. With endpoint admission control, once the connection is started, no further adaptation is needed; by contrast, dynamic adaptation is clearly required for efficient and fair congestion control.

Another major area of related work is the various efforts to short-circuit TCP's slow start delay for moderatesized connections. Typically, these systems use some form of rate pacing for the initial (large) window, but revert to TCP-like behavior for steady state. This allows these systems to be mostly backwardly compatible with existing TCP implementations. As we have argued, however, determining the available bandwidth along a network path is easiest when network traffic is designed to be smooth and well-conditioned. For example, TCP Swift Start [43] uses an initial burst of four packets to measure the physical (not available) capacity of the network path. The hosts then set their initial window to be a fixed fraction (e.g. 1/8th) of the physical capacity. If the bottleneck has significant unused bandwidth, this works great, but it can theoretically create persistent congestion if the rate of arriving flows is greater than the fixed fraction can support. TCP Fast Start [42] and the MIT Congestion Manager [6] use history to guide the selection of the initial congestion window and other TCP parameters; however, their approach only works for nearly simultaneous connections to the same destination.

Similarly, several previous efforts have proposed using delay information, rather than packet loss, to guide congestion control. An early example of this was the packet pair algorithm, using the delay spread from back to back packets to measure the bottleneck bandwidth through a fair-queued router [35]. Packet pair does not perform well with FIFO queues, however, as all endpoints would send at the maximum capacity of the link. Two more recent examples are TCP Vegas [10] and FastTCP [31]. The motivation in each case was to improve steady state TCP performance. As we have argued, many TCP transfers never reach steady state on today's networks.

Finally, we use many individual pieces of technology developed in other contexts. XCP was the first to show that separate mechanisms could be used to provide efficiency and eventual fairness in a congestion control algorithm [32]. In XCP, routers allocate resources to flows without keeping per-flow state. If there is idle capacity, flow rates are rapidly increased without regard to fairness; eventual fairness is provided as a background additive increase/multiplicative decrease process applied

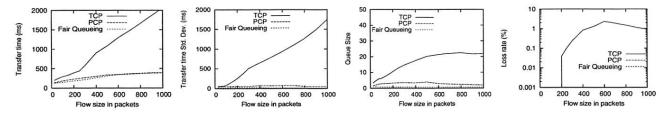


Figure 10: Effect of varying flow size. Bottleneck bandwidth is 40 Mb/s, average RTT is 25 ms, and interarrival times are set to operate the system at 60% load. Fair queueing and PCP suffer no packet loss.

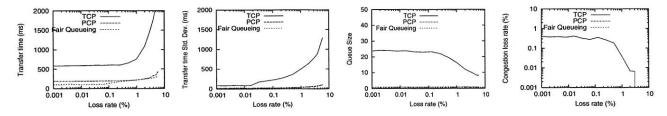


Figure 11: Performance over a lossy channel. Bottleneck bandwidth is 40 Mb/s, average RTT is 25 ms, flow lengths are 250 KB, and interarrival times are set to operate the system at 60% load. Fair queueing and PCP suffer no congestion packet loss.

to all flows. We directly leverage their approach in our work; in fact, we initially started our effort to investigate whether we could achieve the efficiency of XCP without router support. Similarly, Jain and Dovrolis show how to efficiently measure the available bandwidth (capacity less utilization) of an Internet path, by successively probing the path with faster and faster rates until measured delays start increasing [29]. We use a version of their technique, with two principal differences. Their motivation was to measure Internet paths; ours is congestion control. Thus, we carefully meter our probe rates, to ensure that they usually succeed; Jain and Dovrolis attempt to drive the network to the point where the probe starts causing congestion. Their work is also complicated by the fact that TCP traffic is particularly bursty at short time-scales, requiring much longer measurements than in our system. By ensuring that all traffic is paced, we can use shorter and more precise probes. Finally, we note that rate pacing has been proposed as a way to speed TCP startup [42, 43, 28]; if the TCP congestion control variables can be measured or predicted, pacing can provide a way to smooth the initial flight of traffic. We build on this work by using rate pacing throughout the lifetime of a connection, to provide high resource utilization with low delay variance. Earlier work has shown that this form of complete rate pacing interacts poorly with TCP dynamics, by delaying the onset of congestion [1]. In our system, however, the fact that competing flows use rate pacing allows an endpoint to quickly and accurately find if there is available capacity, avoiding the need to drive the resource to overload to determine its resource limits.

6 Conclusion

We have presented the design, implementation and evaluation of PCP, a novel architecture for distributed congestion control with minimal router support. By using short, paced, high-rate bursts, PCP is able to quickly converge to the desired bandwidth in the common case of lightly loaded links with good history. Although our results are somewhat preliminary, we have demonstrated that our approach can significantly outperform TCP, and approach optimal, for response time, loss rate, queue occupancy, and fairness. We believe PCP's combination of techniques shows great promise as a distributed resource allocation mechanism for modern networks.

Acknowledgments

We thank our shepherd Bryan Lyles and other NSDI reviewers for their comments on this paper. We also thank Dave Andersen for providing us access to the wide-area RON testbed.

References

- A. Aggarwal, S. Savage, and T. Anderson. Understanding the Performance of TCP Pacing. In *Proc. IEEE INFOCOM* 2000, Mar. 2000.
- [2] A. Akella, S. Seshan, and A. Shaikh. An Empirical Evaluation of Wide Area Internet Bottlenecks. In *Proc. Internet Measurement Conference*, pages 101–114, Miami Beach, FL, Oct. 2003.
- [3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th SOSP*, 2001.
- [4] The ATM Forum Traffic Management Specification Version 4.0, 1996.
- [5] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A Comparison of Mechanisms for Improving TCP Performance

- over Wireless Links. IEEE/ACM Transactions on Networking, 5(6):756-769, 1997.
- [6] H. Balakrishnan, H. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proc. ACM SIGCOMM '99*, pages 175–187, Boston, MA, Aug. 1999.
- [7] G. Bianchi, A. Capone, and C. Petrioli. Throughput analysis of end-to-end measurement-based admission control in IP. In *Proc.* IEEE INFOCOM 2000, Mar. 2000.
- [8] D. Boggs, J. Mogul, and C. Kent. Measured Capacity of an Ethernet: Myths and Reality. In *Proc. ACM SIGCOMM* '88, Stanford, CA, Aug. 1988.
- [9] B. Braden and alia. Recommendations on Queue Management and Congestion Avoidance in the Internet. IETF RFC-2309, Apr. 1008
- [10] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. In *IEEE Journal on Selected Ar*eas in Communications, volume 13(8), pages 1465–1480, 1995.
- [11] L. Breslau, E. Knightly, S. Shenker, I. Stoica, and H. Zhang. End-point Admission Control: Architectural Issues and Performance. In *Proc. ACM SIGCOMM 2000*, Stockholm, Sweden, Aug. 2000.
- [12] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP Latency. In Proc. IEEE INFOCOM 2000, Mar. 2000.
- [13] D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of ACM SIGCOMM '92*, pages 14–26, 1992.
- [14] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair-Queueing Algorithm. In *Proc. ACM SIGCOMM* '89, 1989
- [15] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor Sharing Flows in the Internet. In Proc. of IWQoS, 2005.
- [16] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. ACM SIGCOMM Computer Communication Review, 36(1), 2006.
- [17] V. Elek, G. Karlsson, and R. Ronngren. Admission control based on end-to-end measurements. In Proc. of IEEE INFOCOM, 2000.
- [18] S. Floyd. TCP and Explicit Congestion Notification. ACM SIG-COMM Computer Communication Review, 24(5):10–23, Oct. 1994
- [19] S. Floyd. HighSpeed TCP for Large Congestion Windows. IETF RFC-3649, Dec. 2003.
- [20] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, Aug. 1999.
- [21] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.
- [22] C. Fraleigh and alia. Packet-level Traffic Measurement from the Sprint IP Backbone. *IEEE Network Magazine*, Nov. 2003.
- [23] R. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Proc. 16th International Teletraffic Congress*, Edinburgh, UK, June 1999.
- [24] R. Gibbens and F. Kelly. On packet marking at priority queues. IEEE Transactions on Automatic Control, 47:1016–1020, 2002.
- [25] J. Gray. Distributed Computing Economics. Technical Report MSR-TR-2003-24, Microsoft Research, Mar. 2003.
- [26] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. IETF RFC-1323, May 1992.
- [27] V. Jacobson and M. Karels. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM* '88, 1988.

- [28] A. Jain and S. Floyd. Quick Start for TCP and IP. IETF Draft, Feb. 2005.
- [29] M. Jain and C. Dovrolis. End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *Proc. ACM SIGCOMM 2002*, Pittsburgh, PA, Aug. 2002.
- [30] C. Jin, D. Wei, and S. Low. The case for delay-based congestion control. In *Proc. IEEE Computer Communication Workshop*, Laguna Beach, CA, Oct. 2003.
- [31] C. Jin, D. Wei, and S. Low. Fast TCP: Motivation, Architecture, Algorithms, Performance. In Proc. IEEE INFOCOM 2004, 2004.
- [32] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIG-COMM* 2002, Pittsburgh, PA, Aug. 2002.
- [33] F. Kelly, P. Key, and S. Zachary. Distributed Admission Control. In *IEEE Journal on Selected Areas in Communications*, 2000.
- [34] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. Computer Communication Review, 2003.
- [35] S. Keshav. A Control-Theoretic Approach to Flow Control. In Proc. ACM SIGCOMM '91, Aug. 1991.
- [36] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level internet path diagnosis. In *Proc. of 19th SOSP*, 2003.
- [37] L. Massoulie and J. Roberts. Arguments in favour of admission control for tcp flows. In *Proc. of ITC 16*, 1999.
- [38] A. Medina, M. Allman, and S. Floyd. Measuring the Evolution of Transport Protocols in the Internet. http://www.icir.org/tbit/TCPevolution-Dec2004.pdf, Dec. 2004.
- [39] R. Morris. TCP Behavior with Many Flows. In Proc. of ICNP, 1997
- [40] V. Oberle and U. Walter. Micro-second precision timer support for the Linux kernel. Technical report, IBM Linux Challenge, Nov. 2001.
- [41] A. Odlyzko. Data Networks are Lightly Utilized, and will Stay that Way. Review of Network Economics, 2(3), Sept. 2003.
- [42] V. Padmanabhan and R. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In Proc. of Globecom Internet Mini-Conf., 1998.
- [43] C. Partridge, D. Rockwell, M. Allman, R. Krishnan, and J. Sterbenz. A Swifter Start of TCP. Technical Report 8339, BBN Tech., 2002.
- [44] K. K. Ramakrishnan and R. Jain. Congestion Avoidance in Computer Networks. Technical Report TR-510, DEC, Aug. 1987.
- [45] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. IEEE INFOCOM 1999*, pages 1337–1345, 1999.
- [46] L. Roberts. Performance of Explicit Rate Flow Control in ATM Networks. In Proc. COMPCON Spring '96, 1996.
- [47] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. ACM Transactions on Computer Systems, 2(4), Nov. 1984.
- [48] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An Analysis of Internet Content Delivery Systems. In *Proc. 5th* OSDI, 2002.
- [49] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. ACM SIG-COMM Computer Communication Review, 29(5):71–78, Oct. 1999.

Availability of Multi-Object Operations

Haifeng Yu
Intel Research Pittsburgh / CMU
yhf@cs.cmu.edu

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

Suman Nath*

Microsoft Research
sumann@microsoft.com

Abstract

Highly-available distributed storage systems are commonly designed to optimize the availability of individual data objects, despite the fact that user level tasks typically request multiple objects. In this paper, we show that the *assignment* of object replicas (or fragments, in the case of erasure coding) to machines plays a dramatic role in the availability of such *multi-object operations*, without affecting the availability of individual objects. For example, for the TPC-H benchmark under real-world failures, we observe differences of up to *four nines* between popular assignments used in existing systems. Experiments using our wide-area storage system prototype, MOAT, on the PlanetLab, as well as extensive simulations, show which assignments lead to the highest availability for a given setting.

1 Introduction

With the fast advance of systems research, performance is no longer the sole focus of systems design [19]. In particular, system availability is quickly gaining importance in both industry and the research community. Data redundancy (i.e., replication or erasure coding) is one of the key approaches for improving system availability. When designing highly-available systems, researchers typically optimize for the availability of individual data objects. For example CFS [9] aims to achieve a certain availability target for individual file blocks, while OceanStore [26] and Glacier [18] focus on the availability of individual (variable-size) objects. However, a user-level task or operation typically requests multiple data objects. For example, in order to compile a project, all of its files need to be available for the compilation to succeed. Similarly, a database query typically requests multiple database objects.

This work is motivated by the following question: Is optimizing the availability of *individual* data objects an effective approach for ensuring the high availability of these multi-object operations? We observe that existing distributed storage systems can differ dramatically in how they assign replicas, relative to each other, to machines. For example, systems such as GFS [15], FARSITE [5], and RIO [33] assign replicas randomly to machines (we call this strategy RAND); others such as the original RAID [28] and Coda [25] manually partition the objects into sets and then mirror each set across multiple machines (we call this strategy PTN); others such as Chord [36] assign replicas to consecutive machines on the DHT ring. However, in spite of the existence of many different assignment strategies, previous studies have not provided general insight across strategies nor have they compared the availability among the strategies for multi-object operations. This leads to the central question of this paper: What is the impact of the replicas' relative assignment on the availability of multiobject operations?

Answering the above two questions is crucial for designing highly-available distributed systems. A negative answer to the first question would suggest that system designers need to think about system availability in a different way—we should optimize availability for multi-object operations instead of simply for individual objects. An answer to the second question would provide valuable design guidelines toward such optimizations.

This paper is the first to study and answer these two questions, using a combination of trace/model-driven simulation and real system deployment. Our results show that, surprisingly, different object assignment strategies result in dramatically different availability for multi-object operations, even though the strategies provide the same availability for individual objects and use the same degree of replication. For example, we observe differences of multiple nines arising between popular assignments used in existing systems such as CAN [29], CFS [9], Chord, Coda, FARSITE, GFS, GHT [30], Glacier [18], Pastry [31], R-CHash [22], RAID, and RIO. In particular, the difference under the TPC-H benchmark reaches four nines: some popular assignments provide less than 50% availability, even

^{*}Work done while this author was a graduate student at CMU and an intern at Intel Research Pittsburgh.

when individual objects have 5 nines availability, while others provide up to 99.97% availability for the same degree of replication.

To answer the second question above, we examine the entire class of possible assignment strategies, including the aforementioned RAND and PTN, in the context of two types of multi-objects operations: strict operations that cannot tolerate any missing objects in the answer (i.e., that require complete answers) and more tolerant operations that are not strict.

We design our simulation experiments based on an initial analytical study of assignment strategies under some specific parameter settings [44]. Our initial analysis [44] indicates that i) for strict operations, PTN provides the best availability while RAND provides the worst; ii) for certain operations that are more tolerant, RAND provides the best availability while PTN provides the worst; and iii) it is impossible to achieve the best of both PTN and RAND.

Based on the above theoretical guidance, we design our simulation study to explore the large parameter space that is not covered by the analysis. Our simulation shows that although operations can have many different tolerance levels for missing objects, as a practical rule of thumb, only two levels matter when selecting an assignment: does the operation require all requested objects (strict) or not (loose)? The results show that the above analytical result for "certain operations that are more tolerant" generalizes to all loose operations. Namely, for all loose operations, RAND tends to provide the best availability while PTN tends to provide the worst. These results have the following implications for multi-object availability: PTNbased systems such as RAID and Coda are optimized for strict operations; RAND-based systems such as GFS, FAR-SITE, and RIO are optimized for loose operations; and other assignment strategies, such as the one used in Chord, lie between PTN and RAND.

Next, we consider practical ways to implement PTN and RAND in distributed systems where objects and machines may be added or deleted over time. CAN approximates RAND in such a dynamic setting. On the other hand, PTN is more challenging to approximate due to its rigid structure. We propose a simple design that approximates PTN in dynamic settings. We have implemented our design for PTN, as well as other assignment strategies, in a prototype wide-area distributed storage system called MOAT (Multi-Object Assignment Toolkit). Although our prototype considers the challenges of wide-area distributed storage, our findings apply to local-area systems as well.

Finally, we study multi-object availability in the presence of two important real-world factors: load imbalance resulting from the use of consistent hashing [22] and correlated machine failures experienced by most wide-area systems [42]. We study these effects using MOAT under a model for network failures, a real eight-month-long Plan-

etLab failure trace, a query trace obtained from the Iris-Log network monitoring system [2], and the TPC-H benchmark. We use both live PlanetLab deployment and event-driven simulation as our testbed. Our results show three intriguing facts. First, both consistent hashing and machine failure correlation hurt the availability of more tolerant operations, but surprisingly, they slightly improve the availability of more strict operations (if the availability of individual objects is kept constant). Second, popular assignments such as Glacier that approximate PTN under perfect load balancing, fail to do so under consistent hashing. Third, our earlier conclusions (which assume perfect load balance and independent machine failures) hold even with consistent hashing and correlated failures: the relative ranking among the assignments remains unchanged.

Although this paper focuses solely on availability, object assignment also affects performance—exploring the interaction between performance and availability goals is part of our future work. Note that in some cases, these goals can be achieved separately, by using a primary storage system for performance goals and a backup storage system (that uses replication or erasure coding) for availability goals [18].

In the next section we discuss motivating applications and examples. Section 3 defines our system model and gives a classification of popular assignments. Section 4 shows that PTN and RAND dominate other assignments. Section 5 presents our designs to approximate PTN in dynamic settings. Section 6 describes MOAT and evaluates assignments under real-world workloads and faultloads. Section 7 discusses related work and Section 8 presents conclusions.

2 Motivation

2.1 Motivating Applications

In most applications including traditional file systems and database systems, user operations tend to request multiple objects. These applications can easily motivate our work. In this section, however, we focus on two classes of applications that are extreme in terms of the number of objects requested by common operations. For each operation, we will focus on the number of objects requested and the tolerance for missing objects.

Image databases. In recent years, computer systems are increasingly being used to store and serve image data [6, 21, 35, 37]. These databases can be quite large. For example, with each 2D protein image object being 4MB, a distributed bio-molecular image database [35] can easily reach multiple terabytes of data. The SkyServer astronomy database [37], which stores the images of astronomical spheres, is rapidly growing, with the potential of generating one petabyte of new data per year [38]. Such large databases are typically distributed among multiple

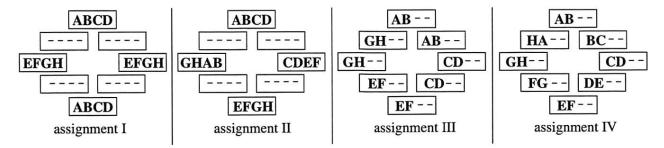


Figure 1: Four possible assignments of 8 objects, **A** through **H**, to 8 machines. Each box represents a machine. A dash (-) indicates an object that is not accessed by the given query.

machines either residing in a LAN or distributed in the wide-area (e.g., similar to the Grid [1]). High availability has been an integral requirement of these systems. For example, the TerraServer system for aerial images explicitly aims for four nines availability [6].

Queries to these image databases can touch a non-trivial fraction of the entire database. For example, among the 35 typical queries used to benchmark SkyServer, at least one query touches over 9.7% of the entire database, while at least four other queries touch over 0.5% of the entire database [17]. Clearly, these queries touch a large number of objects. In other image databases [21], it is difficult to suitably index the data because queries are not known *a priori* and often require domain-specific knowledge. As a result, each query essentially searches every object in the database.

The requirements from these queries can vary based on their semantics. For example, a SkyServer query "compute the average brightness of all galaxies" would likely be able to tolerate some missing objects. On the other hand, a query of "check whether any of the images in the database contain the face of a criminal" would likely require checking all objects in the database, and is thus a strict operation.

Data storage used in network monitoring. Our second class of applications is Internet-scale distributed storage systems such as IrisNet [2, 11, 16], SDIMS [41] and PIER [20] that are used for monitoring potentially hundreds of thousands of network endpoints. In order to avoid the unnecessary bandwidth consumption for data transfer, data are typically stored near their sources (e.g., at the hosts being monitored) [2, 11, 20, 41]. As a result, the database is distributed over a large number of wide-area hosts. Many queries from these applications request aggregated information over a large data set, e.g., the distribution of resource usage over the hosts, the correlation (join) of the worm-infected hosts and their operating systems, etc. Each such guery touches a non-trivial fraction of the entire database. These aggregation queries are likely to be able to tolerate some missing objects. However, other queries, e.g., by a system administrator trying to pinpoint a network problem or find all virus-infected hosts, may not be able to tolerate any missing objects, and are thus strict operations.

2.2 Motivating Example

With our motivating applications in mind, the following simple example illustrates the impact and the subtleties of object assignment.

A simple example with subtle answers. Consider an image database with 16 objects and a query that requests 8 of the 16 objects, namely A through H. An example is the query of "check whether any of the images in the database contain the face of a given male criminal", where A through H are the images with male faces. Because of the nature of this operation, the query is successful only if all the 8 images A–H are available. Suppose each object has exactly two copies, there are 8 identical machines on which we can place these copies, and each machine may hold no more than four objects. Each machine may fail (crash), causing all its data to become unavailable. An object is unavailable if and only if both its copies are unavailable. For simplicity, assume that machines fail independently with the same probability p < 0.5.

Figure 1 gives four (of the many) possible assignments of objects to machines, depicting only the 8 objects requested by the query. Which of these four assignments gives us a better chance that all 8 image objects are available so that the query for criminal faces succeeds? Intuitively, it may make sense that concentrating the objects on fewer machines, as in assignments I and II, gives us a better chance. However, that still leaves a choice between assignment I and assignment II. A careful calculation shows that in fact assignment I provides better availability than assignment II¹, and hence the best availability among all four assignments.

Now consider a network monitoring database with 16 objects, and a query for the average load on U.S. hosts, where objects **A–H** contain the load information for the U.S. hosts. Suppose we are willing to tolerate some error in the average and the query succeeds as long as we can

¹The failure probabilities are $FP(I) = p^4 + 4p^3(1-p) + 2p^2(1-p)^2$ and $FP(II) = p^4 + 4p^3(1-p) + 4p^2(1-p)^2$.

retrieve 5 or more objects. Intuitively, it may now make sense that spreading the objects across more machines, as in assignments III and IV, gives us a better chance that the query succeeds. However, that still leaves a choice between assignments III and IV and again it is not clear which is better. A careful calculation shows that the relative assignment of objects in assignment IV² provides the best availability among all four assignments.

What happens when the query requires 6 or 7 objects to succeed instead of 5 or 8? What about all the other possible assignments that place two objects per machine? Do any of them provide significantly better availability than assignment IV? For databases with millions of objects and hundreds of machines, answering these questions by brute-force calculation is not feasible, so effective guidelines are clearly needed.

Example remains valid under erasure coding. Our simple example uses replication for each object. The exact same problem also arises with erasure coding, where we assign fragments (instead of copies) to machines. If the number of fragments per object is the same as the total number of machines in the system (e.g., 8 in our example), then the assignment problem goes away. However, in large-scale systems, the total number of machines is typically much larger than the number of fragments per object. As a result, the same choice regarding fragment assignment arises.

Also, in our simple example, it is possible to use erasure coding across all objects (i.e., treating them as a single piece of data). This would clearly minimize the failure probability if we need all the objects for the operation to be successful. However, due to the nature of erasure coding, it is not practical to use erasure coding across large amounts of data (e.g., using erasure coding across all the data in the database). Specifically, for queries that request only some of the objects (as in our example), erasure coding across all the objects means that much more data is fetched than is needed for the query. On the other hand, when objects are small, it is practical to use erasure coding across sets of objects. In such scenarios, we view each erasure-coded set of objects as a single logical object. In fact, we intentionally use a relatively large object size of 33MB in some of our later experiments to capture such scenarios.

Summary. The impact of object assignment on availability is complicated and subtle. Intuitive rules make sense, such as "concentrate objects on fewer machines for strict operations" and "spread objects across machines for more tolerant operations". However, these intuitive rules are not useful for selecting among assignments with the same degree of concentration/spread (e.g, for choosing between assignments I and II in our example). This paper provides

N	number of objects in the system
k	number of FORs per object
m	number of FORs needed to reconstruct
	an object (out of the k FORs)
n	number of objects requested by an operation
t	number of objects needed for the operation
	to be successful (out of the n objects)
s	number of machines in the system
l	number of FORs on each machine (= Nk/s)
p	failure probability of each machine
$FP(\alpha)$	failure probability of assignment α

Table 1: *Notation used in this paper.*

effective guidelines for selecting among all assignments, including among assignments with the same degree of concentration/spread. As our results show, such guidelines are crucial: popular assignments with the same degree of concentration/spread can still vary by multiple nines in the availability they provide for multi-object operations. For the example above, our results will show that for the query that cannot tolerate missing objects, assignment I is actually near optimal among all possible assignments. On the other hand, for more tolerant queries, a random assignment of the objects to the machines (with each machine holding two objects) will give us the highest availability.

3 Preliminaries

In this section, we set the context for our work by presenting our system model and then reviewing and classifying well-known assignments.

3.1 System Model

We begin by defining our system model for both replicated and erasure-coded objects. Table 1 summarizes the notation we use.

There are N data *objects* in the system, where an object is, for example, a file block, a file, a database tuple, a group of database tuples, an image, etc. An operation requests (for reading and/or writing) n objects, $1 \le n \le N$, to perform a certain user-level task. There are s machines in the system, each of which may experience crash (benign) failures with a certain probability p. Replication or erasure coding is used to provide fault tolerance. Each object has k replicas (for replication) or k fragments (for erasure coding). We use the same k for all objects to ensure a minimal level of fault tolerance for each object. Extending the model and our results to different k's is part of our future work. To unify terminology, we call each fragment or replica a FOR of the object. The k FORs of an object are numbered 1 through k. We assume that m out of kFORs are needed for a given object to be available for use.

 $^{^2{\}rm The}$ failure probabilities are $FP(III)=p^8+8p^7(1-p)+28p^6(1-p)^2+24p^5(1-p)^3+6p^4(1-p)^4$ and $FP(IV)=p^8+8p^7(1-p)+28p^6(1-p)^2+8p^5(1-p)^3.$

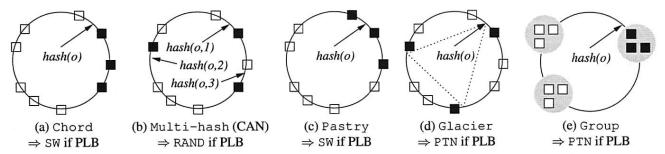


Figure 2: Placement of a single object o in different consistent hashing-based assignments used in various systems. The machines (shown as squares) have random IDs in the circular ID space. The object is replicated on three machines (shown as black solid squares). Each single object placement rule determines a different relative placement among objects, which in turn results in different availability. For each such assignment, we also note the corresponding ideal assignment if consistent hashing achieved perfect load balancing (PLB).

non-ideal assignments (consistent-hashing-based)	systems using similar non-ideal assignments	
Chord [36] (Figure 2(a)): i th successor of $hash(o)$		
Multi-hash (Figure 2(b)): 1st successor of $hash(o, i)$	CAN [29], GFS [15], FARSITE [5], RIO [33]	
Pastry [31] (Figure 2(c)): machine with the <i>i</i> th closest ID to $hash(o)$		
Glacier [18] (Figure 2(d)): 1st successor of $hash(o) + MAXID \cdot (i-1)/k$	GHT [30], R-CHash [22, 40]	
Group (Figure 2(e)): See Section 5.1	Original RAID [28], Coda [25, 34] ³	

Table 2: Salient object assignments. For the assignments in the first column, we note to which machine the ith FOR of object o is assigned.

If an object has less then m FORs available, we say the object fails. A global assignment (or simply assignment) is a mapping from the kN FORs to the s machines in the system. An assignment is ideal (in terms of load balance) if each machine has exactly l = kN/s FORs. The value l is also called the load of a machine.

For an operation requesting n objects, if not all n objects are available, the operation may or may not be considered successful, depending on its tolerance for missing objects. This paper studies threshold criteria: an operation is successful if and only if at least t out of the n objects are available. Here t is an operation-specific value from 1 to n based on the application semantics.

We need to emphasize that the operation threshold t is not to be confused with the m in m-out-of-k erasure coding:

- In erasure coding, a single object is encoded into k
 fragments, and we can reconstruct the object from
 any m fragments. Moreover, the reconstructed object
 is the same regardless of which m fragments are retrieved.
- For an operation with a threshold t, it does not reconstruct the n objects. Rather, the user may be reasonably satisfied even if only t objects are retrieved because of the specific application semantics. Depending on which t objects are retrieved, the answer to the

user query may be different. But the user is willing to accept any of these approximate answers.

Finally, we define the availability of an operation as the probability that it is successful. We use "number of nines" (i.e., $\log_{0.1}(1-availability)$) to describe availability. The complement of availability is called unavailability or failure probability. For a given operation, we use $FP(\alpha)$ to denote the failure probability of a particular assignment α . When we say that one assignment α is x nines better than another assignment β , we mean $\log_{0.1} FP(\alpha) - \log_{0.1} FP(\beta) = x$. Finally, our availability definition currently does not capture possible retries or the notion of "wait time" before a failed operation can succeed by retrying. We intend to investigate these aspects in our future work.

3.2 Classifying Well-Known Assignments

Next, we review popular assignments from the literature, and then define three ideal assignments.

We focus on well-known assignments based on consistent hashing [22]. In consistent hashing, each machine has a numerical ID (between 0 and MAXID) obtained by, for example, pseudo-randomly hashing its own IP address. All machines are organized into a single ring where the machine IDs are non-decreasing clockwise along the ring (except at the point where the ID space wraps around).

Figure 2 visualizes and Table 2 describes the assignments used in Chord, CAN, Pastry and Glacier. Intuitively, in Chord, the object is hashed once and then as-

³Note that Coda [25] itself does not restrict the assignment from volumes to servers. However, in most Coda deployments [34], system administrators use an assignment similar to PTN.

signed to the k successors of the hash value. In CAN (or Multi-hash), the object is hashed k times using k different hash functions, and assigned to the k immediate successors of the k hash results. Pastry also hashes the object, but it assigns the object to the machines with the k closest IDs to the hash value. Finally, Glacier hashes the object and then places the object at k equi-distant points on the ID ring. Because of the use of consistent hashing, machines in these assignments may not have exactly the same load; hence, by definition, the assignments are not ideal. Table 2 also lists other popular assignments that are similar to the ones discussed above.

Next we define three ideal assignments. RAND is the assignment obtained by randomly permuting the Nk FORs and then assigning the permutation to the machines (l FORs per machine) sequentially. Note that strictly speaking, RAND is a distribution of assignments. If the machine IDs and the hashes of the objects in consistent hashing were exactly evenly distributed around the ring, then Multi-hash would be the same as RAND (Figure 2(b)). In PTN, we partition objects into sets and then mirror each set across multiple machines. Specifically, the FORs of lobjects are assigned to machines 1 through k, the FORs of another l objects are assigned to machines k+1 through 2k, and so on. If consistent hashing provided perfect load balancing, then Glacier would be the same as PTN (Figure 2(d)). This is because all objects whose hashes fall into the three ID regions (delimited by black solid squares and their corresponding predecessors) will be placed on the three black solid squares, and those three machines will not host any other objects. Finally, in SW (sliding window), the FORs of l/k objects are assigned to machines 1 through k, the FORs of another l/k objects are assigned to machines 2 through k+1, and so on. If consistent hashing provided perfect load balancing, then Chord and Pastry would be the same as SW (Figures 2(a) and (c)), because all objects falling within the ID region between a machine and its predecessor will be assigned to the same k successors.

Finally, we define the concept of a projected assignment for a given operation. For an assignment and a given operation requesting n objects, the projected assignment is the mapping from the nk FORs of the n objects to the machines. In other words, in the projected assignment, we ignore objects not requested by the operation. We extend the definitions of PTN and RAND to projected assignments. A projected assignment is called PTN if the global assignment is PTN and the nk FORs reside on exactly nk/l machines. Namely, the n objects should concentrate on as few machines as possible and obey the PTN rule within those machines (as in assignment I of Figure 1, where n=8 and

k=2).⁵ Similarly, a projected assignment is called RAND if the global assignment is RAND and the nk FORs reside on exactly $\min(nk,s)$ machines. Here, RAND spreads the n objects on as many machines as possible. In Figure 1, assignments III and IV have the desired spread, but such well-structured assignments are highly unlikely to occur under the RAND rule. When the context is clear, we will not explicitly distinguish an assignment from its projected assignment.

4 Study of Ideal Assignments

In this section, we investigate the ideal assignments under independent machine failures. Later, Section 6 will study the more practical assignments under real failure traces.

4.1 Simulation of Ideal Assignments

We begin our study by using simulation to compare RAND, PTN and SW. We consider here the case where n=N and leave the cases for n< N to our later evaluation of practical assignments. There are six free parameters in our simulation: N, s, k, m, p and t. We have performed a thorough simulation over the entire parameter space and considered additional assignments beyond those in Section 3.2, but in this paper we are able to present only a small subset of our results (Figure 3). Each of the observations described below extends to all the other parameter settings and assignments with which we experimented. Note that Figure 3 and other figures in this paper use a relatively large failure probability p, in order to show the underlying trend with confidence, given the limited duration of our simulations. The observations and conclusions do not change with smaller p.

Figure 3 shows that when t = n, PTN has the lowest unavailability (roughly 0.08) among the three assignments in the figure. In contrast, when t = n RAND has the highest unavailability (nearly 1) among the three. Hence, PTN is the best and RAND is the worst when t = n. As t decreases, the unavailability of PTN does not change until we are able to tolerate 300 missing objects (i.e., $t/n \approx 98.7\%$). The reason is that in PTN, whenever one object is unavailable, then the other l-1 objects on the same set of machines become unavailable as well (l = 300). For RAND, the unavailability decreases much faster as we are able to tolerate more and more missing objects. The curves for PTN and RAND cross when $t/n \approx 99.8\%$, below which point RAND becomes the best among all assignments while PTN roughly becomes the worst. This crossing point appears to be not far from the availability probability for individual objects, i.e., $1 - p^k \approx 99.9\%$. When t/n = 99.4%, the difference between PTN and RAND is already three nines.

⁴Strictly speaking, CAN uses consistent hashing in multiple dimensions instead of a single dimension. Thus we use the term Multi-hash to describe this assignment.

⁵Note that assignments II and III fail to have the PTN rule and the desired concentration, respectively.

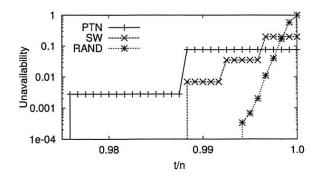


Figure 3: Unavailability of ideal assignments for an operation that requests all 24000 objects stored on 240 machines in the system. The number of machines is set to match our PlanetLab deployment. Each object has 3 replicas, and each machine fails with probability 0.1. The x-axis is the fraction (t/n) of the 24000 objects that needs to be available for the operation to succeed.

The intuition behind the above results is that each assignment has a certain amount of "inter-object correlation". Because each machine may hold FORs of multiple objects, these objects become correlated even if machine failures are independent. Intuitively, PTN is the assignment that maximizes inter-object correlation, while RAND minimizes it. When t is very close to n, larger inter-object correlation is better because it does not help for a small number of objects to be available by themselves. On the other hand, if t is not close to n, smaller inter-object correlation is better because it decreases the chance that many objects fail together.

It is important to note that the crossing between PTN and RAND occurs very close to 100%. As we mentioned earlier, in all our experiments, the crossing point occurs when t/n is near the availability of individual objects. As long as this availability is reasonably high, the crossing point will be close to 100%. This observation has significant practical importance. Namely, despite the fact that t can range from 1 to n, we can largely classify operations into "strict" operations and "loose" operations, as follows: An operation is strict if it cannot tolerate any missing objects, otherwise it is loose. With practical parameters, loose operations will most likely fall into the region where RAND is the best. On the other hand, PTN is best for strict operations.

4.2 Analytical Study of Ideal Assignments

The above simulation study shows that among the assignments we simulated, PTN and RAND are each the best in two different regions. But is this because we missed out on some other assignments? Do we need to consider additional assignments? Definitive answers to these questions are not readily obtained experimentally, because there are exponentially many possible assignments.

We have separately obtained analytical results [44] on optimal assignments under some specific t values and assuming failure independence. Because these results are only for restricted parameter settings and are not the contribution of this paper, following we provide only a brief summary of the analytical results from [44]:⁶

- For t = n (i.e., strict operations), PTN is the best (to within 0.25 nines) and RAND is the worst (to within 0.70 nines) among all possible ideal assignments.
- For t = l + 1 and n = N (or t = 1 and n < N),
 PTN is the worst and RAND is the best (to within 0.31 nines) among all possible ideal assignments.
- It is impossible to achieve the best of both PTN and RAND.

The analysis in [44] also finds a rigorous mathematical definition for inter-object correlation, which confirms our earlier intuition.

5 Designs to Approximate Optimal Assignments

Our study in the previous section shows that PTN and RAND are (near) optimal for strict and loose operations, respectively. This motivates the exploration of practical designs that approximate these ideal assignments when objects and machines may be added or deleted on the fly. Our goal is to approximate not only PTN and RAND, but also their projected assignments for n < N. We have also explored optimizing solutions for systems where strict and loose operations may coexist. For lack of space, we defer the solutions to [43]. We refer the reader back to Table 2 for definitions of various non-ideal assignments.

RAND is already approximated by Multi-hash. Moreover, for any operation requesting n objects, Multi-hash is likely to spread the nk FORs evenly on the ID ring. This means that the projected assignment will also approximate RAND. Thus, we do not need any further design in order to approximate RAND.

For PTN, RAID [28] and Coda [34, 25] achieve PTN by considering only a static set of machines (or disks in the case of RAID). Adding or deleting machines requires human intervention. Glacier handles the dynamic case, and it would have achieved PTN if consistent hashing provided perfect load balancing. However, we will see later that in practice, it behaves similar to Chord (and hence far from PTN). Therefore, we propose a *Group DHT* (or Group in short) design that better approximates PTN. Regardless of whether we use Glacier, Chord, Pastry

 $^{^6} Because$ it only wastes resources for one machine to host multiple FORs of the same object, we consider only assignments where each machine has ≤ 1 FOR of any given object. The only exception is RAND, where some assignments in the distribution may violate this property.

or Group, their projected assignments will not approximate PTN when n < N. Therefore, we further propose designs to ensure that the projected assignments approximate PTN for n < N.

Our designs are compatible with the standard DHT routing mechanisms for locating objects. It is worth pointing out that when n is large, DHT routing will be inefficient. For those cases, multicast techniques such as in PIER [20] can be used to retrieve the objects. Our designs are compatible with those techniques as well. Finally, for cases where a centralized directory server is feasible (e.g., in a LAN cluster), neither DHT routing nor multicast techniques are required for our design.

5.1 Approximating PTN for n = N

This section describes how we approach PTN with Group DHT. The design itself is not the main contribution or focus of this paper – thus we will provide only a brief description, and leave the analysis of Group DHT's performance, as well as discussions of security issues, to [43].

Basic Group DHT design. In Group DHT (or Group), each DHT node is a group of exactly k machines (Figure 2(e) provides an example for k=3). We assign the k FORs of an object to the k machines in the successor group of the object's hash value. Here we assume that all objects have the same number of FORs, and a more general design is part of our future work. There is a small number r (e.g., r=s/1000) of "rendezvous" machines in the system that help us form groups.

For machine join, it is crucial to observe that a machine joins the system for two separate purposes: using the DHT (as a client) and providing service (as a server). A machine can always use the DHT by utilizing some other existing machine (that is already in the DHT) as a proxy, even before itself becomes part of the ring. It must be able to find such a proxy because it needs to know a bootstrap point to join the DHT.

In order to provide service to other machines, a machine first registers with a random rendezvous. If there are less than k new machines registered with the rendezvous at this point, the new machine simply waits. Otherwise, the k new machines form a group, and join the DHT ring. During the delayed join, the new machine can still use the DHT as a client – it simply cannot contribute. The only factor we need to consider then is whether there will be a large fraction of machines that cannot contribute. With 1/1000 of the machines serving as rendezvous machines, each with at most k-1 waiting, the fraction of the machines that are waiting is at most (k-1)/1000. Given that k is a small number such as 5, this means that only 0.4% of the ma-

chines in the system are not being utilized, which is negligible.

When a machine in a group fails or departs, the group has two options. The first option would be to dismiss itself entirely, and then have the k-1 remaining machines join the DHT again. This may result in thrashing because the leave/join rate is artificially inflated by a factor of k. The second option would be for the group to wait, and hope to recruit a new machine so that it can recover to k machines. However, doing so causes some objects to have fewer than k FORs for possibly an extended period of time.

In our design, we use a mixture of both options. When a group loses a member, it registers with a random rendezvous. If the rendezvous has a new machine registered with it, the group will recruit the new machine as its member. If the group is not able to recruit a new machine before the total number of members drops from k-1 to k-2, it dismisses itself. The threshold of k-2 is tunable, and a smaller value will decrease the join/leave rate at the cost of having fewer replicas on average. However, our study shows that even a threshold of k-2 yields a near optimal join/leave rate, and hence we always use k-2 as the threshold. Finally, the group will also dismiss itself if it has waited longer than a certain threshold amount of time.

Rendezvous. It is important to remember that the rendezvous machines are contacted only upon machine join and leave, and not during object retrieval/lookup. In our system, we intend to maintain roughly r=s/1000 rendezvous in the group DHT. This r is well above the number of machines needed to sustain the load incurred by machine join/leave under practical settings, and yet small enough to keep the fraction of un-utilized machines negligible.

We use the following design to dynamically increase/decrease r with s. Each group independently becomes a rendezvous with probability of 1/1000. These rendezvous then use the Redir [24] protocol to form a smaller rendezvous DHT. To contact a random rendezvous, a machine simply chooses a random key and searches for the successor of the key in the smaller rendezvous DHT. As with other groups in the system, rendezvous groups may fail or leave. Fortunately, the states maintained by rendezvous groups are soft states, and we simply use periodic refresh.

5.2 Approximating PTN for n < N

Group approximates a global PTN assignment. However, for an operation requesting n < N objects, the corresponding projected assignment will not be PTN. This is because the hash function spreads the n objects around the ring, whereas the projected PTN assignment requires the n objects to occupy as few machines as possible. Next we present designs for approximating projected PTN, using known designs for supporting range queries.

 $^{^7}$ In this section, we use *node* to denote a logical node in the DHT and *machine* to denote one of the s physical machines.

Defining a global ordering. To ensure that the projected assignments approximate PTN, we first define an ordering among all the objects. The ordering should be such that most operations roughly request a "range" of objects according to the ordering. Note that the operations need not be real range queries. In many applications, objects are semantically organized into a tree and operations tends to request entire subtrees. For example, in network monitoring systems, users tends to ask aggregation questions regarding some particular regions in the network. In the case of file systems, if a user requests one block in a file, she will likely request the entire file. Similarly, files in the same directories are likely to be requested together. For these hierarchical objects, we can easily use the full path from the root to the object as its name, and the order is directly defined alphabetically by object names.

Placing objects on the ID ring according to the order. After defining a global ordering among the objects, we use an order-preserving hash function [14] to generate the IDs of the objects. Compared to a standard hash function, for a given ordering "<" among the objects, an order-preserving hash function $hash_{order}()$ has the extra guarantee that if $o_1 < o_2$, then $hash_{order}(o_1) < hash_{order}(o_2)$. If we have some knowledge regarding the distribution of the object names (e.g., when the objects are names in a telephone directory), then it is possible [14] to make the hash function uniform as well. The "uniform" guarantee is important because it ensures the load balancing achieved by consistent hashing. Otherwise some ID regions may have more objects than others.

For cases where a uniform order-preserving function is not possible to construct, we further adopt designs [7, 23] for supporting range queries in DHTs. In particular, MOAT uses the item-balancing DHT [23] design to achieve dynamic load balancing. Item-balancing DHT is the same as Chord except that each node periodically contacts a random other node in the system to adjust load (without disturbing the order).

Finally, there are also cases where a single order cannot be defined over the objects. We are currently investigating how to address those cases using database clustering algorithms [46].

6 Study of Practical Assignments

In this section, we use our MOAT prototype, real failure traces, and real workloads to study consistent hashing-based assignments. In particular, we will answer the following two questions that were not answered in Section 4: Which assignment is the best under the effects of imperfect load balancing in consistent hashing, and also under the effects of machine failure correlation? How do the results change from our earlier study on ideal assignments?

For lack of space, we will consider in this section only the scenario where each object has 3 replicas, unless otherwise noted. We have also performed extensive experiments for general erasure coding with different m and k values—the results we obtain are qualitatively similar and all our claims in this section still hold. In the following, we will first describe our MOAT prototype, the failure traces and the workload, and then thoroughly study consistent hashing-based assignments.

6.1 MOAT Implementation

We have incorporated the designs in the previous section into a read/write wide-area distributed storage system called MOAT. MOAT is similar to PAST [32], except that it supports all the consistent-hashing-based assignments discussed in this paper. Specifically, it supports Glacier, Chord, Group and Multi-hash. For Group, unless otherwise mentioned, we mean Group with the ordering technique from Section 5.2. Other assignments do not use the ordering technique. MOAT currently only supports optimistic (best effort) consistency. We have implemented MOAT by modifying FreePastry 1.3.2 [13]. MOAT is written in Java 1.4, and uses nonblocking I/O and Java serialization for communication.

Despite the fact that we support DHT routing in MOAT, as we mentioned in Section 5, DHT routing will not be used if either a centralized server is feasible or when the number of objects requested by an operation is large. To isolate DHT routing failures (i.e., failures by the DHT to locate an object) from object failures and to better focus on the effects of assignments, in all our experiments we define availability as the probability that some live, accessible machine in the system has that object.

6.2 Faultloads and Workloads

A *faultload* is, intuitively, a failure workload, and describes when failures occur and on which machines. We consider two different faultloads intended to capture two different failure scenarios. The first faultload, *NetModel*, is a synthetic one and aims to capture short-term machine unavailability caused by local network failures that partition the local machine from the rest of the network, rendering it inaccessible. We use the network failure model from [10] with a MTTF of 11571 seconds, MTTR of 609 seconds, and a failure probability of p=0.05. The MTTR is directly from [10], while the MTTF is calculated from our choice of p.

The second faultload, *PLtrace*, is a pair-wise ping trace [3] among over 200 PlanetLab machines from April 2004 to November 2004. Because of the relatively large

⁸In the remainder of the paper, we will not explicitly discuss Pastry as it is similar to Chord for the purposes of this paper.

(15 minutes) sampling interval, PLtrace mainly captures machine failures rather than network failures. This trace enables us to study the effects of failure correlation, FOR repair (i.e., generating new FORs to compensate for lost FORs due to machine failure), as well as heterogeneous machine failure probabilities. In our evaluation, we consider a machine to have failed if *none* of the other machines can ping it. Further details about how we process the trace can be found in [43].

We also use two real workloads for user operations, the TPC-H benchmark and a query log from IrisLog [2]. Our two workloads are intended to represent the two classes of applications in Section 2. Note that TPC-H is not actually a benchmark for image databases. However, it has a similar data-mining nature and most queries touch a large number of objects (e.g., several touch over 5% of the database).

In our TPC-H workload, we use an 800GB database (i.e., a TPC-H scaling factor of 8000) and 240 MOAT machines. Because of our 3-fold replication overhead, the overall database size is 2.4TB.9 Each object is roughly 33MB and contains around 29,000 consecutive tuples in the relational tables. Note that we intentionally use a relatively large object size to take into account the potential effects of erasure coding across smaller objects (recall Section 2.2). Using smaller objects sizes will only further increase the differences among the assignments and magnify the importance of using the appropriate assignments. The ordering among the objects for TPC-H is defined to be the ordering determined by (table name, tuple name), where tuple name is the primary key of the first tuple in the object. Note that most queries in TPC-H are not actually range queries on the primary key. So this enables us to study the effect of a non-ideal ordering among objects.

In our IrisLog workload, the query trace contains 6,467 queries processed by the system from 11/2003 to 8/2004. IrisLog maintains 3530 objects that correspond to the load, bandwidth, and other information about PlanetLab machines. The number of objects requested by each query ranges from 10 to 3530, with an average of 704. The objects in IrisLog form a logical tree based on domain names of the machines being monitored. The ordering among the objects is simply defined according to their full path from the root of the tree. In contrast to TPC-H, the operations in IrisLog request contiguous ranges of objects along the ordering.

6.3 Effects of Consistent Hashing

We perform this set of experiments by deploying the 240 MOAT machines on 80 PlanetLab machines and using the network failure faultload of NetModel. The machines span

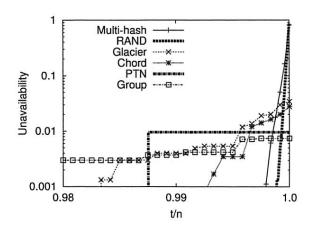


Figure 4: Unavailability of an operation requesting all 24,000 objects in the system, under the NetModel faultload. t/n is the fraction of objects that needs to be available for the operation to succeed.

North America, Europe and Asia, and include both academic and non-academic sites. Each machine emulates locally whether it is experiencing a network failure according to NetModel, and logs the availability of its local objects. We then measure the availability of different operations by collecting and analyzing the logs. During the experiments, there were no failures caused by PlanetLab itself, and all failures experienced by the system were emulated failures. For Group, we use only a single rendezvous node.

Figure 4 plots the unavailability of a single operation requesting all 24,000 objects in MOAT under Multi-hash, Glacier, Chord and Group. We focus on t values that are close to n, to highlight the crossing points between assignments. We also obtained three curves (via simulation as in Section 4.1) for their counterpart ideal assignments (i.e., PTN, SW, and RAND). For clarity, however, we omit the SW curve. The general trends indicate that our earlier claims about the optimality of PTN and RAND hold for Group and Multi-hash. Furthermore, the crossing point between PTN and RAND is rather close to n.

The same conclusion holds for Figure 5, which plots the unavailability of a much smaller "range" operation (requesting only 600 objects). The large difference among different assignments shows that object assignments have dramatic availability impact even on operations that request a relatively small number of objects. The 600 objects requested only comprise of 2.5% of the 24,000 objects in the system. It is also easy to observe that for n < N, the order-preserving hash function (in Group with order) is necessary to ensure good availability. Next we look at two deeper aspects of these plots.

Does consistent hashing increase or decrease availability? In Figure 4, Group is close to PTN, which means it well-approximates the optimal assignment of

⁹For comparison, industrial vendors use a 10 TB database with TPC-H in clusters with 160 machines [4].

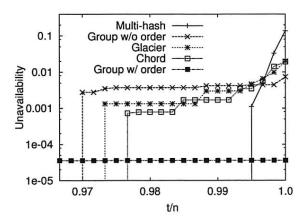


Figure 5: Unavailability of a smaller operation that requests only 600 objects (among the 24,000 object in the system), under the NetModel faultload. Again, t/n is the fraction of the 600 objects that needs to be available for the operation to succeed.

PTN. Multi-hash has the same trend as RAND but it does depart from RAND significantly. The imperfect load balancing in consistent hashing decreases the slope of the Multi-hash curve (as compared to RAND), and makes it slightly closer to PTN. This means that the unavailability for loose operations is increased, while the unavailability for strict operations is decreased. We also observe similar effects of consistent hashing when comparing Chord against SW, and Group against PTN (except that the effects are much smaller).

We pointed out in Section 4 that the key difference between PTN and RAND is that PTN maximizes the inter-object correlation while RAND minimizes the inter-object correlation. Imperfect load balancing (as in Group and Multi-hash) increases such inter-object correlation. As a result, consistent hashing makes all curves closer to PTN.

We argue that this effect from imperfect load balancing in consistent hashing should be explicitly taken into account in systems design, because the difference between RAND and Multi-hash can easily reach multiple nines. For example, when t/n=99.8% in Figure 4, the unavailability under Multi-hash is 1.12×10^{-3} , while the unavailability under RAND is only 1.67×10^{-5} (not shown in the graph).

Does Glacier approximate PTN well? As with Group, the counterpart ideal assignment for Glacier is PTN, the best assignment for strict operations. However, Figure 4 clearly shows that Glacier is much closer to Chord than to Group when t/n is close to 1.0. This can be explained by carefully investigating the inter-object correlation in these assignments and counting the number of machines intersecting with any given machine. Two machines intersect if they host FORs from the same objects.

A smaller number of intersecting machines (as in PTN) roughly indicates larger inter-object correlation.

In Chord, the total number of intersecting machines is roughly 2(k-1), where k is the number of replicas or fragments per object. In Group, this number is k. For Glacier, suppose that the given machine is responsible for ID region (v_1,v_2) . The next set of FORs for the objects in this region will fall within $(v_1 + \text{MAXID}/k, v_2 + \text{MAXID}/k)$. Unless this region exactly falls within a machine boundary, it will be split across two machines. Following this logic, the total number of intersecting machines is roughly 2(k-1). This explains why Glacier is closer to Chord than to Group when t/n is close to 1.0.

When t=n in Figure 4, the unavailability of Chord (0.027) and Glacier (0.034) is about 4 times that of Group (0.0074). The advantage of Group becomes larger when k increases. We observe in our experiments that when using the NetModel faultload with p=0.2 and 12-out-of-24 erasure coding (i.e., m=12 and k=24), the unavailability of Chord and Glacier is 0.0117 and 0.0147, respectively. On the other hand, Group has an unavailability of only 0.00067 – roughly a 20-fold advantage. In our other experiments we also consistently observe that the difference between Group and Glacier is about k times.

6.4 Effects of Failure Correlation

We next use our second faultload, PLtrace, to study the effects of correlated machine failures (together with consistent hashing). Given that we want to obtain a fair comparison across different assignments, we need the system to observe only failures injected according to the traces and not the (non-deterministic) failures on the PlanetLab. This is rather unlikely using our live PlanetLab deployment given our eight month long trace and the required duration of the experiments.

Simulation validation via real deployment. To solve this problem, we use a mixture of real deployment and event-driven simulation for PLtrace. Using trace compression, we are able to inject and replay a one-week portion of PLtrace into our MOAT deployment on the PlanetLab in around 12 hours. To observe a sufficient number of failures, we intentionally choose a week that has a relatively large number of failures. These 12-hour experiments are repeated many times (around 20 times) to obtain two runs (one for Group and one for Multi-hash) without non-deterministic failures.

These two successful runs then serve as validation points for our event-driven simulator. We feed the same one-week trace into our event-driven simulator and then compare the results to validate its accuracy (Figure 6). It is easy to see that the two simulation curves almost exactly match the two curves from the PlanetLab deployment, which means our simulator has satisfactory accuracy. For space reasons,

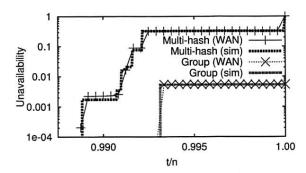


Figure 6: Validation results: Unavailability of a single operation that requests all 24,000 objects in the system, as measured in WAN deployment and as observed in event-driven simulation. The number of machines and the machine failure probability are determined by PLtrace. We cannot observe any probability below 10^{-4} due to the duration of the experiment.

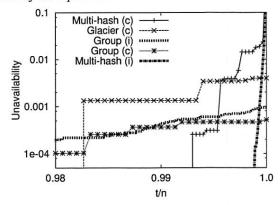


Figure 7: Effects of failure correlation: The availability of an operation that requests all 24,000 objects in the system. The legend "(c)" means that the curve is for PLtrace with correlated failures, while "(i)" means that the curve is obtained assuming independent failures.

we omit other validation results. We next inject the entire PLtrace into our simulator.

Does failure correlation increase or decrease availability? Figure 7 plots the unavailability of a single operation under PLtrace for Glacier, Multi-hash, and Group. For clarity, we did not plot the curve for Chord, which is close to Glacier. The data (not included in Figure 7) show that for all three settings, the average unavailability of individual objects is around 10^{-5} . We then perform a separate simulation assuming that each machine fails independently with a failure probability of 0.0215, a value chosen such that $0.0215^k \approx 10^{-5}$ (recall k=3). We include the corresponding simulation curves (for the three assignments) under this independent failure model in Figure 7 as well. For clarity, we omit the curve for Glacier under independent failures.

Comparing the two sets of curves reveals that ma-

chine failure correlation makes all the curves move toward Group and away from Multi-hash (i.e., decreasing the slope of the curves). The effect is the most prominent when we compare Multi-hash(c) and Multi-hash(i). In retrospect, this phenomenon is easy to explain. The reason is exactly the same as the effect of imperfect load balancing discussed in Section 6.3. Namely, machine failure correlation increases the inter-object correlation of all assignments. This also provides intuition regarding why our earlier conclusions (assuming failure independence) on the optimality of PTN and RAND hold even under correlated failures. Namely, even though all assignments become closer to PTN under correlated failures, their relative ranking will not change because the extra "amount" of correlation added is the same for all assignments.

6.5 Overall Availability for Real Workloads

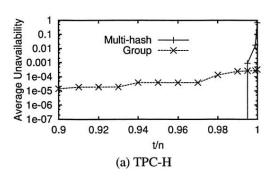
Up to this point, we have presented results only for the availability of individual operations. This section investigates the overall availability of all operations in our two real workloads, via simulation driven by PLtrace. Because the queries in the workloads are of different sizes, here we assume that they have the same t/n values. These results provide a realistic view of how much availability improvement we can achieve by using appropriate assignments.

The TPC-H benchmark consists of 22 different queries on a given database. To evaluate their availability in our simulator, we first instantiate the given database as a MySQL database and use it to process the queries. We then record the set of tuples touched by each query. Finally, we simulate a replicated and distributed TPC-H database and use the trace to determine the availability of each query.

We plot only two assignments because our results so far have already shown that Group and Multi-hash are each near-optimal in different regions. In both Figure 8(a) and (b), we see that when t=n, Group outperforms Multi-hash by almost 4 nines. On the other hand, for t/n=90%, Multi-hash achieves at least 2 more nines than Group; this difference becomes even larger when t/n<90%. The absolute availability achieved under the two workloads are different largely due to the different sizes of the operations. In TPC-H, the operations request more objects than in the IrisLog trace. Finally, the crossing between Group and RAND again occurs when t is quite close to n. This indicates that from a practical perspective, we may consider only whether an operation is able to tolerate missing objects, rather than its specific t.

7 Related Work

To the best of our knowledge, this paper is the first to study the effects of object assignment on multi-object operation availability. On the surface, object assignment is



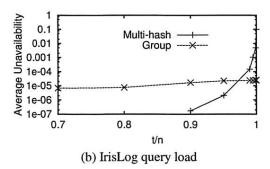


Figure 8: Overall availability under real workloads and the PLtrace faultload.

related to replica placement. Replica placement has been extensively studied for both performance and availability. Replica placement research for availability [8, 12, 45] typically considers the availability of individual objects rather than multi-object operations. These previous results on replica placement cannot be easily extended to multi-object operations because the two problems are fundamentally different. For example, the replica placement problem goes away if all the machines are identical, but as our results show, assignment still affects availability even when all the machines are identical.

Despite the fact that previous systems [5, 15, 22, 29, 31, 36, 39] use different assignments for objects, all systems except Chain Replication [39] study only the performance (rather than the availability) of object assignments (if the effects of assignment are explored at all). Chain replication [39] investigates the availability of individual objects in LAN environments. In their setting, the availability of individual objects is influenced by the different data repair times for different assignments. For example, after a machine failure, in order to restore (repair) the replication degree for the objects on that failed machine, it is faster to create new replicas of these objects on many different target machines in parallel. As a result, more randomized assignments such as Multi-hash are preferable to more structured assignments such as Group. Compared to Chain Replication, this paper studies the effects of assignments on multi-object operations. The findings from Chain Replication and this paper are orthogonal. For example, for strict operations, our study shows that Group yields much higher availability than Multi-hash. When restoring lost replicas, we can still use the pattern as suggested in Chain Replication, and temporarily restore the replicas of the objects on many different machines. The object replicas can then be lazily moved to the appropriate places as determined by the desired assignment (e.g., by Group). In this way, all assignments will enjoy the same minimum repair time.

As in our Group design, the concept of grouping nodes is also used in Viceroy [27], but for a different purpose of bounding the in-degrees of nodes. Because of the different

purpose, the size of each group in Viceroy can vary between $c_1 \log s$ to $c_2 \log s$, where c_1 and c_2 are constants. Viceroy maintains the groups simply by splitting a group if it is too large, or merging a group with its adjacent group if it is too small. In our design, the group sizes have less variance, and we achieve this by the use of rendezvous.

This paper studies the effects of object assignments experimentally. In [44], we have also obtained initial theoretical optimality results under some specific parameter settings (namely, when t=n and t=l+1). Using experimental methods, this paper answers the object assignment question for all t values. It also investigates the effects of two real-world factors—failure correlation and imperfect load balancing—that were not considered in [44].

8 Conclusion

This paper is the first to reveal the importance of object assignment to the availability of multi-object operations. Without affecting the availability of individual objects or resource usage, appropriate assignments can easily improve the availability of multi-object operations by multiple nines. We show that under realistic parameters, if an operation cannot tolerate missing objects, then PTN is the best assignment while RAND is the worst. Otherwise RAND is the best while PTN is the worst. Designs to approximate these assignments, Multi-hash and Group, respectively, have been implemented in MOAT and evaluated on the PlanetLab. Our results show differences of 2-4 nines between Group and Multi-hash for both an IrisLog query trace and the TPC-H benchmark.

References

- [1] Grid. http://www.grid.org.
- [2] IrisLog: A Distributed SysLog. http://www.intel-iris. net/irislog.
- [3] PlanetLab All Pair Pings. http://www.pdos.lcs.mit.edu/~strib/pl_app/.
- [4] Top Ten TPC-H by Performance. http://www.tpc.org/ tpch/results/tpch_perf_results.asp.

- [5] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In OSDI (2002).
- [6] BARCLAY, T., AND GRAY, J. Terraserver san-cluster architecture and operations experience. Tech. Rep. MSR-TR-2004-67, Microsoft Research. 2004.
- [7] BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. Mercury: Supporting Scalable Multi-Attribute Range Queries. In SIGCOMM (2004).
- [8] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In SIGMETRICS (2000).
- [9] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In ACM SOSP (2001).
- [10] DAHLIN, M., CHANDRA, B., GAO, L., AND NAYATE, A. End-to-end WAN Service Availability. ACM/IEEE Transactions on Networking 11, 2 (April 2003).
- [11] DESHPANDE, A., NATH, S., GIBBONS, P. B., AND SESHAN, S. Cache-and-query for Wide Area Sensor Databases. In ACM SIG-MOD (2003).
- [12] DOUCEUR, J. R., AND WATTENHOFER, R. P. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In DISC (2001).
- [13] FREEPASTRY. http://www.cs.rice.edu/CS/Systems/ Pastry/FreePastry.
- [14] GARG, A., AND GOTLIEB, C. Order-Preserving Key Transformations. ACM Transactions on Database Systems 11, 2 (June 1986), 213–234.
- [15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In ACM SOSP (2003).
- [16] GIBBONS, P. B., KARP, B., KE, Y., NATH, S., AND SESHAN, S. IrisNet: An Architecture for a Worldwide Sensor Web. *IEEE Pervasive Computing* 2, 4 (2003).
- [17] GRAY, J., SLUTZ, D., SZALAY, A., THAKAR, A., VANDENBERG, J., KUNSZT, P., AND STOUGHTON, C. Data Mining the SDSS SkyServer Database. Tech. Rep. MSR-TR-2002-01, Microsoft Research, January 2002.
- [18] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In NSDI (2005).
- [19] HENNESSY, J. The Future of Systems Research. *IEEE Computer* 32, 8 (August 1999), 27–33.
- [20] HUEBSCH, R., HELLERSTEIN, J. M., BOON, N. L., LOO, T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In VLDB (2003).
- [21] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G., RIEDEL, E., AND AIL-AMAKI., A. Diamond: A storage architecture for early discard in interactive search. In USENIX (FAST) (2004).
- [22] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In ACM Symposium on Theory of Computing (May 1997)
- [23] KARGER, D., AND RUHL, M. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In ACM SPAA (2004).
- [24] KARP, B., RATNASAMY, S., RHEA, S., AND SHENKER, S. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *IPTPS* (2004).

- [25] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems 10, 1 (Feb. 1992), 3–25.
- [26] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-scale Persistent Storage. In ACM ASPLOS (2000).
- [27] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In ACM PODC (2002)
- [28] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In ACM SIGMOD (1988).
- [29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-addressable Network. In ACM SIGCOMM (2001).
- [30] RATNASAMY, S., KARP, B., SHENKER, S., ESTRIN, D., GOVIN-DAN, R., YIN, L., AND YU, F. Data-Centric Storage in Sensornets with GHT, A Geographic Hash Table. *Mobile Networks and Appli*cations (MONET) 8, 4 (August 2003).
- [31] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In ACM Middleware (2001).
- [32] ROWSTRON, A., AND DRUSCHEL, P. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In ACM SOSP (2001).
- [33] SANTOS, J., MUNTZ, R., AND RIBEIRO-NETO, B. Comparing Random Data Allocation and Data Striping in Multimedia Serviers. In ACM SIGMETRICS (2000).
- [34] SATYANARAYANAN, M. Private communication, 2005.
- [35] SINGH, A. K., MANJUNATH, B. S., AND MURPHY, R. F. A Distributed Database for Bio-molecular Images. SIGMOD Rec. 33, 2 (2004), 65–71.
- [36] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In ACM SIGCOMM (2001).
- [37] SZALAY, A., KUNSZT, P., THAKAR, A., GRAY, J., AND SLUTZ, D. Designing and Mining MultiTerabyte Astronomy Archives: The Sloan Digital Sky Survey. In ACM SIGMOD (June 1999).
- [38] SZALAY, A. S., GRAY, J., AND VANDENBERG, J. Petabyte Scale Data Mining: Dream or Reality? Tech. Rep. MSR-TR-2002-84, Microsoft Research, August 2002.
- [39] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain Replication for Supporting High Throughput and Availability. In OSDI (2004).
- [40] WANG, L., PAI, V., AND PETERSON, L. The Effectiveness of Request Redirection on CDN Robustness. In OSDI (2002).
- [41] YALAGANDULA, P., AND DAHLIN, M. A scalable distributed information management system. In ACM SIGCOMM (2004).
- [42] YALAGANDULA, P., NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Large Distributed Systems. In WORLDS (2004).
- [43] YU, H., GIBBONS, P. B., AND NATH, S. K. Availability of Multi-Object Operations. Tech. rep., Intel Research Pittsburgh, 2005. Technical Report IRP-TR-05-53. Also available at http: //www.cs.cmu.edu/~yhf/moattr.pdf.
- [44] YU, H., GIBBONS, P. B., AND NATH, S. K. Optimal-Availability Inter-Object Correlation. Tech. rep., Intel Research Pittsburgh, 2006. Technical Report IRP-TR-06-03. Also available at http://www.cs.cmu.edu/~yhf/interobject.pdf.
- [45] YU, H., AND VAHDAT, A. Minimal Replication Cost for Availability. In ACM PODC (2002).
- [46] ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In ACM SIGMOD (1996).

Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems

Suman Nath*
Microsoft Research
sumann@microsoft.com

Haifeng Yu Phillip B. Gibbons

Intel Research Pittsburgh

{haifeng.yu,phillip.b.gibbons}@intel.com

Srinivasan Seshan
Carnegie Mellon University
srini@cmu.edu

Abstract

High availability is widely accepted as an explicit requirement for distributed storage systems. Tolerating correlated failures is a key issue in achieving high availability in today's wide-area environments. This paper systematically revisits previously proposed techniques for addressing correlated failures. Using several real-world failure traces, we qualitatively answer four important questions regarding how to design systems to tolerate such failures. Based on our results, we identify a set of design principles that system builders can use to tolerate correlated failures. We show how these lessons can be effectively used by incorporating them into IRISSTORE, a distributed read-write storage layer that provides high availability. Our results using IRISSTORE on the PlanetLab over an 8-month period demonstrate its ability to withstand large correlated failures and meet preconfigured availability targets.

1 Introduction

High availability is widely accepted as an explicit requirement for distributed storage systems (e.g., [8, 9, 10, 14, 15, 19, 22, 41]). This is partly because these systems are often used to store important data, and partly because performance is no longer the primary limiting factor in the utility of distributed storage systems. Under the assumption of failure independence, replicating or erasure coding the data provides an effective way to mask individual node failures. In fact, most distributed storage systems today use some form of replication or erasure coding.

In reality, the assumption of failure independence is rarely true. Node failures are often correlated, with multiple nodes in the system failing (nearly) simultaneously. The size of these correlated failures can be quite large. For example, Akamai experienced large distributed denial-of-service (DDoS) attacks on its servers in May and June 2004 that resulted in the unavailability of many of its client sites [1]. The PlanetLab experienced four failure events during the first half of 2004 in which more than 35 nodes

(≈ 20%) failed within a few minutes. Such large correlated failure events may have numerous causes, including system software bugs, DDoS attacks, virus/worm infections, node overload, and human errors. The impact of failure correlation on system unavailability is dramatic (i.e., by orders of magnitude) [6, 38]. As a result, tolerating correlated failures is a key issue in designing highly-available distributed storage systems. Even though researchers have long been aware of correlated failures, most systems [8, 9, 10, 14, 15, 40, 41] are still evaluated and compared under the assumption of independent failures.

In the context of the IrisNet project [17], we aim to design and implement a distributed read/write storage layer that provides high availability despite correlated node failures in non-P2P wide-area environments (such as the PlanetLab and Akamai). We study three real-world failure traces for such environments to evaluate the impact of realistic failure patterns on system designs. We frame our study around providing quantitative answers to several important questions, some of which involve the effectiveness of existing approaches [19, 37]. We show that some existing approaches, although plausible, are less effective than one might hope under real-world failure correlation, often resulting in system designs that are far from optimal. Our study also reveals the subtleties that cause this discrepancy between the expected behavior and the reality. These new findings lead to four design principles for tolerating correlated failures. While some of the findings are perhaps less surprising than others, none of the findings and design principles were explicitly identified or carefully quantified prior to our work. These design principles are applied and implemented in our highly-available read/write storage layer called IRISSTORE.

Our study answers the following four questions about tolerating correlated failures:

Can correlated failures be avoided by history-based failure pattern prediction? We find that avoiding correlated failures by predicting the failure pattern based on externally-observed failure history (as proposed in OceanStore [37]) provides *negligible* benefits in alleviating the negative effects of correlated failures in our real-world

^{*}Work done while this author was a graduate student at CMU and an intern at Intel Research Pittsburgh.

failure traces. The subtle reason is that the top 1% of correlated failures (in terms of size) have a dominant effect on system availability, and their failure patterns seem to be the most difficult to predict.

Is simple modeling of failure sizes adequate? We find that considering only a single (maximum) failure size (as in Glacier [19]) leads to suboptimal system designs. Under the same level of failure correlation, the system configuration as obtained in [19] can be both overly-pessimistic for lower availability targets (thereby wasting resources) and overly-optimistic for higher availability targets (thereby missing the targets). While it is obvious that simplifying assumptions (such as assuming a single failure size) will always introduce inaccuracy, our contribution is to show that the inaccuracy can be dramatic in practical settings. For example, in our traces, assuming a single failure size leads to designs that either waste 2.4 times the needed resources or miss the availability target by 2 nines. Hence, more careful modeling is crucial. We propose using a bi-exponential model to capture the distribution of failure sizes, and show how this helps avoid overly-optimistic or overly-pessimistic designs.

Are additional fragments/replicas always effective in improving availability? For popular (n/2)-out-of-n encoding schemes (used in OceanStore [22] and CFS [10]), as well as majority voting schemes [35] over n replicas, it is well known that increasing n yields an exponential decrease in unavailability under independent failures. In contrast, we find that under real-world failure traces with correlated failures, additional fragments/replicas result in a strongly diminishing return in availability improvement for many schemes including the previous two. It is important to note that this diminishing return does not directly result from the simple presence of correlated failures. For example, we observe no diminishing return under Glacier's single-failure-size correlation model. Rather, in our realworld failure traces, the diminishing return arises due to the combined impact of correlated failures of different sizes. We further observe that the diminishing return effects are so strong that even doubling or tripling n provides only limited benefits after a certain point.

Do superior designs under independent failures remain superior under correlated failures? We find that the effects of failure correlation on different designs are dramatically different. Thus selecting between two designs \mathcal{D} and \mathcal{D}' based on their availability under independent failures may lead to the wrong choice: \mathcal{D} can be far superior under independent failures but far inferior under real-world correlated failures. For example, our results show that while 8-out-of-16 encoding achieves 1.5 more nines of availability than 1-out-of-4 encoding under independent failures, it achieves 2 fewer nines than 1-out-of-4 encoding under correlated failures. Thus, system designs must be explicitly evaluated under correlated failures.

Our findings, unavoidably, depend on the failure traces we used. Among the four findings, the first one may be the most dependent on the specific traces. The other three findings, on the other hand, are likely to hold as long as failure correlation is non-trivial and has a wide range of failure sizes.

We have incorporated these lessons and design principles into our IRISSTORE prototype: IRISSTORE does not try to avoid correlated failures by predicting the correlation pattern; it uses a failure size distribution model rather than a single failure size; and it explicitly quantifies and compares configurations via online simulation with our correlation model. As an example application, we built a publicly-available distributed wide-area network monitoring system on top of IRISSTORE, and deployed it on over 450 Planet-Lab nodes for 8 months. We summarize our performance and availability experience with IRISSTORE, reporting its behavior during a highly unstable period of the PlanetLab (right before the SOSP 2005 conference deadline), and demonstrating its ability to reach a pre-configured availability target.

2 Background and Related Work

Distributed storage systems and erasure coding. Distributed storage systems [8, 9, 10, 14, 15, 19, 22, 41] have long been an active area in systems research. Of these systems, only OceanStore [22] and Glacier [19] explicitly consider correlated failures. OceanStore uses a distributed hash table (DHT) based design to support a significantly larger user population (e.g., 10^{10} users) than previous systems. Glacier is a more robust version of the PAST [32] DHT-based storage system that is explicitly designed to tolerate correlated failures.

Distributed storage systems commonly use data redundancy (replication, erasure coding [31]) to provide high availability. In erasure coding, a data object is encoded into n fragments, out of which any m fragments can reconstruct the object. OceanStore uses (n/2)-out-of-n erasure coding. This configuration of erasure coding is also used by the most recent version of CFS [10, 13], one of the first DHT-based, read-only storage systems. Glacier, on the other hand, adapts the settings of m and n to achieve availability and resource targets. Replication can be viewed as a special case of erasure coding where m=1.

In large-scale systems, it is often desirable to automatically create new fragments upon the loss of existing ones (due to node failures). We call such systems *regeneration* systems. Almost all recent distributed storage systems (including OceanStore, CFS, and our IRISSTORE system) are regeneration systems. Under the assumption of failure independence, the availability of regeneration systems is typically analyzed [41] using a Markov chain to model the birth-death process.

Trace	Duration	Nature of nodes	# of nodes	Probe interval	Probe method
PL_trace[3]	03/2003 to 06/2004	PlanetLab nodes	277 on avg	15 to 20 mins	all-pair pings; 10 ping packets per probe
WS_trace[6]	09/2001 to 12/2001	Public web servers	130	10 mins	HTTP GET from a CMU machine
RON_trace [4]	03/2003 to 10/2004	RON testbed	30 on avg	1 to 2 mins	all-pair pings; 1 ping packet per probe

Table 1: Three traces used in our study.

Previous availability studies of correlated failures. Traditionally, researchers assume failure independence when studying availability [8, 9, 10, 14, 15, 40, 41]. For storage systems, correlated failures have recently drawn more attention in the context of wide-area environments [11, 19, 21, 37, 38], local-area and campus network environments [9, 34], and disk failures [6, 12]. None of these previous studies point out the findings and design principles in this paper or quantify their effects. Below, we focus on those works most relevant to our study that were not discussed in the previous section.

The Phoenix Recovery Service [21] proposes placing replicas on nodes running heterogeneous versions of software, to better guard against correlated failures. Because their target environment is a heterogeneous environment such as a peer-to-peer system, their approach does not conflict with our findings.

The effects of correlated failures on erasure coding systems have also been studied [6] in the context of survivable storage disks. The study is based on availability traces of desktops [9] and different Web servers. (We use the same Web server trace in our study.) However, because the target context in [6] is disk drives, the study considers much smaller-scale systems (at most 10 disks) than we do.

Finally, this paper establishes a bi-exponential model to fit correlated failure size distribution. Related to our work, Nurmi et al. [30] show that machine availability (instead of failure sizes) in enterprise and wide-area distributed systems also follows hyper-exponential (a more general version of our bi-exponential model) models.

Data maintenance costs. In addition to availability, failure correlation also impacts data maintenance costs (i.e., different amounts of data transferred over the network in regenerating the objects). Weatherspoon et al. [36] show that the maintenance costs of random replica placement (with small optimizations) are similar to those of an idealized optimal placement where failure correlation is completely avoided. These results do not conflict with our conclusion that random replica placement (or even more sophisticated placement) cannot effectively alleviate the negative effects of failure correlation on availability.

3 Methodology

This study is based on system implementation and experimental evaluation. The experiments use a combination of three testbeds: live PlanetLab deployment, realtime emulation on Emulab [2], and event-driven simulation. Each testbed allows progressively more extensive evaluation than the previous one. Moreover, the results from one testbed help validate the results from the next We report availability and performance results from the 8-month live deployment of our system over the PlanetLab. We also present detailed simulation results for deeper understanding into system availability, especially when exploring design choices. Simulation allows us to directly compare different system configurations by subjecting them to identical streams of failure/recovery events, which is not possible using our live deployment. We have carefully validated the accuracy of our simulation results by comparing them against the results from the PlanetLab deployment and also Emulab emulation. Our detailed comparison in [29] shows that there is negligible differences between the three sets of results.

In this paper, we use ERASURE(m, n) to denote an m-out-of-n read-only erasure coding system. Also, to unify terminology, we often refer to replicas as fragments of an ERASURE(1, n) system. Unless otherwise mentioned, all designs we discuss in this paper use regeneration to compensate for lost fragments due to node failures.

For the purpose of studying correlated failures, a *failure event* (or simply *failure*) crashes one or more nodes in the system. The number of nodes that crash is called the *size* of the failure. To distinguish a failure event from the failures of individual nodes, we explicitly call the latter *node failures*. A data object is *unavailable* if it can not be reconstructed due to failures. We present availability results using standard "number of nines" terminology (i.e., $\log_{0.1}(\phi)$, where ϕ is the probability that the data object is unavailable).

Failure traces. We use three real-world wide-area failure traces (Table 1) in our study. WS_trace is intended to be representative of public-access machines that are maintained by different administrative domains, while PL_trace and RON_trace potentially describe the behavior of a centrally administered distributed system that is used mainly for research purposes, as well as for a few long running services.

A probe *interval* is a complete round of all pair-pings or Web server probes. PL_trace¹ and RON_trace consist of periodic probes between every pair of nodes. Each probe may consist of multiple pings; we declare that a node has

¹Note that PL_trace has sporadic "gaps", see [29] for how we classify the gaps and treat them properly based on their causes.

failed if none of the other nodes can ping it during that interval. We do not distinguish between whether the node has failed or has simply been partitioned from all other nodes—in either case it is unavailable to the overall system. WS_trace contains logs of HTTP GET requests from a single node at CMU to multiple Web servers. Our evaluation of this trace is not as precise because near-source network partitions make it appear as if all the other nodes have failed. To mitigate this effect, we assume that the probing node is disconnected from the network if 4 or more consecutive HTTP requests to different servers fail.² We then ignore all failures during that probe period. This heuristic may still not perfectly classify source and server failures, but we believe that the error is likely to be minimal.

In studying correlated failures, each probe interval is considered as a separate failure event whose size is the number of failed nodes that were available during the previous interval. More details on our trace processing methodology can be found in [29].

Limitations. Although the findings from this paper depend on the traces we use, the effects observed in this study are likely to hold as long as failure correlation is non-trivial and has a wide range of failure sizes. One possible exception is our observation about the difficulty in predicting failure patterns of larger failures. However, we believe that the sources of some large failures (e.g., DDoS attacks) make accurate prediction difficult in any deployment.

Because we target IRISSTORE's design for non-P2P environments, we intentionally study failure traces from systems with largely homogeneous software (operating systems, etc.). Even in WS_trace, over 78% of the web servers were using the same Apache server running over Unix, according to the historical data at http://www.netcraft.com. We believe that widearea non-P2P systems (such as Akamai) will largely be homogeneous because of the prohibitive overhead of maintaining multiple versions of software. Failure correlation in P2P systems can be dramatically different from other wide-area systems, because many failures in P2P systems are caused by user departures. In the future, we plan to extend our study to P2P environments. Another limitation of our traces is that the long probe intervals prevent the detection of short-lived failures. This makes our availability results slightly optimistic.

Steps in our study. Section 4 constructs a tunable failure correlation model from our three failure traces; this model allows us to study the sensitivity of our findings beyond the three traces. Sections 5–8 present the questions we study, and their corresponding answers, overlooked subtleties, and design principles. These sections focus on

read-only ERASURE(m,n) systems, and mainly use tracedriven simulation, supplemented by model-driven simulation for sensitivity study. Section 9 shows that our conclusions readily extend to read/write systems. Section 10 describes and evaluates IRISSTORE, including its availability running on PlanetLab for an 8-month period.

4 A Tunable Model for Correlated Failures

This section constructs a tunable failure correlation model from the three failure traces. The primary purpose of this model is to allow sensitivity studies and experiments with correlation levels that are stronger or weaker than in the traces. In addition, the model later enables us to avoid overly-pessimistic and overly-optimistic designs. The model balances idealized assumptions (e.g., Poisson arrival of correlated failures) with realistic characteristics (e.g., mean-time-to-failure and failure size distribution) extracted from the real-world traces. In particular, it aims to accurately capture large (but rare) correlated failures, which have a dominant effect on system unavailability.

4.1 Correlated Failures in Real Traces

We start by investigating the failure correlation in our three traces. Figure 1 plots the PDF of failure event sizes for the three traces. Because of the finite length of the traces, we cannot observe events with probability less than 10^{-5} or 10^{-6} . While RON_trace and WS_trace have a roughly constant node count over their entire duration, there is a large variation in the total number of nodes in PL_trace. To compensate, we use both raw and normalized failure event sizes for PL_trace. The normalized size is the (raw) size multiplied by a normalization factor γ , where γ is the average number of nodes (i.e., 277) in PL_trace divided by the number of nodes in the interval.

In all traces, Figure 1 shows that failure correlation has different strengths in two regions. In PL_trace, for example, the transition between the two regions occurs around event size 10. In both regions, the probability decreases roughly exponentially with the event size (note the log-scale of the y-axis). However, the probability decreases significantly faster for small-scale correlated failures than for large-scale ones. We call such a distribution bi-exponential. Although we have only anecdotal evidence, we conjecture that different failure causes are responsible for the different parts of the distribution. For example, we believe that system instability, some application bugs, and localized network partitions are responsible for the small failure events. It is imaginable that the probability decreases quickly as the scale of the failure increases. On the other hand, human interference, attacks, viruses/worms and large ISP failures are likely to be responsible for the large failures. This is supported by the fact that many of

 $^{^2}$ This threshold is the smallest to provide a plausible number of near-source partitions. Using a smaller threshold would imply that the client (on Internet2) experiences a near-source partition > 4% of the time in our trace, which is rather unlikely.

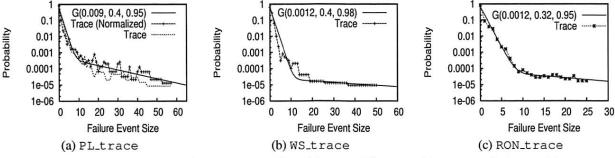


Figure 1: Correlated failures in three real-world traces. $G(\alpha, \rho_1, \rho_2)$ is our correlation model.

the larger PlanetLab failures can be attributed to DDoS attacks (e.g., on 12/17/03), system software bugs (e.g., on 3/17/04), and node overloads (e.g., on 5/14/04). Once such a problem reaches a certain scale, extending its scope is not much harder. Thus, the probability decreases relatively slowly as the scale of the failure increases.

4.2 A Tunable Bi-Exponential Model

In our basic failure model, failure events arrive at the system according to a Poisson distribution. The entire system has a universe of u nodes. The model does not explicitly specify which of the u nodes each failure event crashes, for the following reasons: Predicting failure patterns is not an effective means for improving availability, and patternaware fragment placement achieves almost identical availability as a pattern-oblivious random placement (see Section 5). Thus, our availability study needs to consider only the case where the m fragments of a data object are placed on a random set of m nodes. Such a random placement, in turn, is equivalent to using a fixed set of m nodes and having each failure event (with size s) crash a random set of s nodes in the universe. Realizing the above point helps us to avoid the unnecessary complexity of modeling which nodes each failure event crashes - each failure event simply crashes a random set of s nodes.

We find that many existing distributions (such as heavy-tail distributions [24]) cannot capture the bi-exponential property in the trace. Instead, we use a model that has two exponential components, one for each region. Each component has a tunable parameter ρ between 0 and ∞ that intuitively captures the slope of the curve and controls how strong the correlations are. When $\rho=0$, failures are independent, while $\rho=\infty$ means that every failure event causes the failure of all u nodes. Specifically, for $0<\rho<\infty$, we define the following geometric sequence: $f(\rho,i)=c(\rho)\cdot \rho^i$. The normalizing factor $c(\rho)$ serves to make $\sum_{i=0}^u f(\rho,i)=1$.

We can now easily capture the bi-exponential property by composing two $f(\rho, i)$. Let p_i be the probability of failure events of size i, for $0 \le i \le u$. Our correlation model, denoted as $G(\alpha, \rho_1, \rho_2)$, defines $p_i = (1 - \alpha)f(\rho_1, i) +$ $\alpha f(\rho_2,i)$, where α is a tunable parameter that describes the probability of large-scale correlated failures. Compared to a piece-wise function with different ρ 's for the two regions, $G(\alpha,\rho_1,\rho_2)$ avoids a sharp turning point at the boundary between the two regions.

Figure 1 shows how well this model fits the three traces. The parameters of the models are chosen such that the Root Mean Square errors between the models and the trace points are minimized. Because of space limitations, we are only able to provide a brief comparison among the traces here. The parameters of the model are different across the traces, in large part because the traces have different universe sizes (10 node failures out of 277 nodes is quite different from 10 node failures out of 30 nodes). As an example of a more fair comparison, we selected 130 random nodes from PL_trace to enable a comparison with WS_trace, which has 130 nodes. The resulting trace is well-modeled by G(0.009, 0.3, 0.96). This means that the probability of large-scale correlated failures in PL_trace is about 8 times larger than WS_trace.

Failure arrival rate and recovery. So far the correlation model $G(\alpha, \rho_1, \rho_2)$ only describes the failure event size distribution, but does not specify the event arrival rate. To study the effects of different levels of correlation, the event arrival rate should be such that the average mean-time-to-failure (MTTF) of nodes in the traces is always preserved. Otherwise with a constant failure event arrival rate, increasing the correlation level would have the strong side effect of decreasing node MTTFs. To preserve the MTTF, we determine the system-wide failure event arrival rate λ to be such that $1/(\lambda \sum_{i=1}^{u} (ip_i)) = \text{MTTF}/u$.

We observe from the traces that, in fact, there exists non-trivial correlation among node recoveries as well. Moreover, nodes in the traces have non-uniform MTTR and MTTF. However, our experiments show that the above two factors have little impact on system availability for our study. Specifically, for all parameters we tested (e.g., later in Figure 4), the availability obtained under model-driven simulation (which assumes independent recoveries and uniform MTTF/MTTR) is almost identical to that obtained under trace-driven simulation (which has recovery correlation and non-uniform MTTF/MTTR). There-

fore, our model avoids the unnecessary complexity of modeling recovery correlation and non-uniform MTTF/MTTR.

Stability of the model. We have performed an extensive stability study for our model using PL_trace and RON_trace. We do not use WS_trace due to its short length. We summarize our results below – see [29] for the full results. Our results show that each model converges within a 4 month period in its respective failure traces. We believe this convergence time is quite good given the rarity of large correlation events. Second, the model built ("trained") using a prefix of a trace (e.g., the year 2003 portion) reflects well the failures occurring in the rest of the trace (e.g., the year 2004 portion). This is important because we later use the model to configure IRISSTORE to provide a given availability target.

In the next four sections, we will address each of the four questions from Section 1. Unless otherwise stated, all our results are obtained via event-driven simulation (including data regeneration) based on the three real failure traces. The bi-exponential model is used only when we need to tune the correlation level—in such cases we will explicitly mention its use. As mentioned earlier, the accuracy of our simulator has been carefully validated against live deployment results on the PlanetLab and also emulation results on Emulab. Because many existing systems (e.g., OceanStore, Glacier, and CFS) use a large number of fragments, we show results for up to n=60 fragments (as well as for large values of m).

5 Can Correlated Failures be Avoided by History-based Failure Pattern Prediction?

Chun et al. [11] point out that externally observed node failure histories (based on probing) can be used to discover a relatively stable pattern of correlated failures (i.e., which set of nodes tend to fail together), based on a portion of PL_trace. Weatherspoon et al. [37] (as part of the OceanStore project [22]) reach a similar conclusion by analyzing a four-week failure trace of 306 web servers³. Based on such predictability, they further propose a framework for online monitoring and clustering of nodes. Nodes within the same cluster are highly correlated, while nodes in different clusters are more independent. They show that the clusters constructed from the first two weeks of their trace are similar to those constructed from the last two weeks. Given the stability of the clusters, they conjecture that by placing the n fragments (or replicas) in n different clusters, the n fragments will not observe excessive failure correlation among themselves. In some sense, the problem of correlated failures goes away.

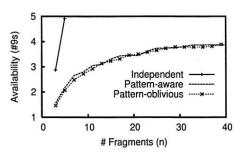


Figure 2: Negligible availability improvements from failure pattern prediction of WS_trace for ERASURE(n/2, n) systems.

Our Finding. We carefully quantify the effectiveness of this technique using the same method as in [37] to process our failure traces. For each trace, we use the first half of the trace (i.e., "training data") to cluster the nodes using the same clustering algorithm [33] as used in [37]. Then, as a case study, we consider two placement schemes of an ERASURE(n/2, n) (as in OceanStore) system. The first scheme (pattern-aware) explicitly places the n fragments of the same object in n different clusters, while the second scheme (pattern-oblivious) simply places the fragments on n random nodes. Finally, we measure the availability under the second half of the trace.

We first observe that most ($\approx 99\%$) of the failure events in the second half of the traces affect only a very small number (≤ 3) of the clusters computed from the first half of the trace. This implies that the clustering of correlated nodes is relatively stable over the two halves of the traces, which is consistent with [37].

On the other hand, Figure 2 plots the achieved availability of the two placement schemes under WS_trace. PL_trace produces similar results [29]. We do not use RON_trace because it contains too few nodes. The graph shows that explicitly choosing different clusters to place the fragments gives us negligible improvement on availability. We also plot the availability achieved if the failures in WS_trace were independent. This is done via modeldriven simulation and by setting the parameters in our biexponential model accordingly. Note that under independent failures, the two placement schemes are identical. The large differences between the curve for independent failures and the other curves show that there are strong negative effects from failure correlation in the trace. Identifying and exploiting failure patterns, however, has almost no effect in alleviating such impacts. We have also obtained similar findings under a wide-range of other m and n values for ERASURE(m, n).

A natural question is whether the above findings are because our traces are different from the traces studied in [11, 37]. Our PL_trace is, in fact, a superset of the failure trace used in [11]. On the other hand, the failure trace studied in [37], which we call Private_trace, is not pub-

³The trace actually contains 1909 web servers, but the authors analyzed only 306 servers because those are the only ones that ever failed during the four weeks.

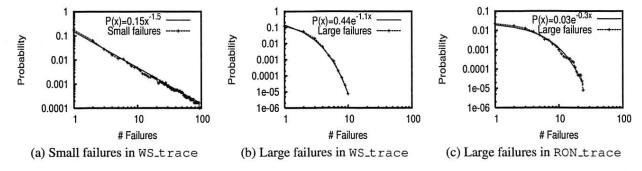


Figure 3: Predictability of pairwise failures. For two nodes selected at random, the plots show the probability that the pair fail together (in correlation) more than x times during the entire trace. Distributions fitting the curves are also shown.

licly available. Is it possible that Private_trace gives even better failure pattern predictability than our traces, so that pattern-aware placement would indeed be effective? To answer this question, we directly compare the "pattern predictability" of the traces using the metric from [37]: the average mutual information among the clusters (MI) (see [37] for a rigorous definition). A smaller MI means that the clusters constructed from the training data predict the failure patterns in the rest of the trace better. Weatherspoon et al. report an MI of 0.7928 for their Private_trace. On the other hand, the MI for our WS_trace is 0.7612. This means that the failure patterns in WS_trace are actually more "predictable" than in Private_trace.

The Subtlety. To understand the above seemingly contradictory observations, we take a deeper look at the failure patterns. To illustrate, we classify the failures into small failures and large failures based on whether the failure size exceeds 15. With this classification, in all traces, most ($\approx 99\%$) of the failures are small.

Next, we investigate how accurately we can predict the failure patterns for the two classes of failures. We use the same approach [20] used for showing that UNIX processes running for a long time are more likely to continue to run for a long time. For a random pair of nodes in WS_trace, Figure 3(a) plots the probability that the pair crashes together (in correlation) more than x times because of the small failures. The straight line in log-log scale indicates that the data fits the Pareto distribution of P(#failure > $x = cx^{-k}$, in this case for k = 1.5. We observe similar fits for PL_trace $(P(\#failure \ge x) = 0.3x^{-1.45})$ and RON_trace $(P(\#failure \ge x) = 0.02x^{-1.4})$. Such a fit to the Pareto distribution implies that pairs of nodes that have failed together many times in the past are more likely to fail together in the future [20]. Therefore, past pairwise failure patterns (and hence the clustering) caused by the small failures are likely to hold in the future.

Next, we move on to large failure events (Figure 3(b)). Here, the data fit an exponential distribution of $P(\#failure \ge x) = ae^{-bx}$. The memoryless property of the exponential distribution means that the frequency

of future correlated failures of two nodes is independent of how often they failed together in the past. This in turn means that we cannot easily (at least using existing history-based approaches) predict the failure patterns caused by these large failure events. Intuitively, large failures are generally caused by external events (e.g., DDoS attacks), occurrences of which are not predictable in trivial ways. We observe similar results [29] in the other two traces. For example, Figure 3(c) shows an exponential distribution fit for the large failure events in RON_trace. For PL_trace, the fit is $P(\#failure \geq x) = 0.3e^{-0.9x}$.

Thus, the pattern for roughly 99% of the failure events (i.e., the small failure events) is predictable, while the pattern for the remaining 1% (i.e., the large failure events) is not easily predictable. On the other hand, our experiments show that large failure events, even though they are only 1% of all failure events, contribute the most to unavailability. For example, the unavailability of a "patternaware" ERASURE(16, 32) under WS_trace remains unchanged (0.0003) even if we remove all the small failure events. The reason is that because small failure events affect only a small number of nodes, they can be almost completely masked by data redundancy and regeneration. This explains why on the one hand, failure patterns are largely predictable, while on the other hand, pattern-aware fragment placement is not effective in improving availability. It is also worth pointing out that the sizes of these large failures still span a wide range (e.g., from 15 to over 50 in PL_trace and WS_trace). So capturing the distribution over all failure sizes is still important in the bi-exponential model.

The Design Principle. Large correlated failures (which comprise a small fraction of all failures) have a dominant effect on system availability, and system designs must not overlook these failures. For example, failure pattern prediction techniques that work well for the bulk of correlated failures but fail for large correlated failures are not effective in alleviating the negative effects of correlated failures.

Can Correlated Failures be Avoided by Root Causebased Failure Pattern Prediction? We have established

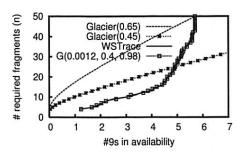


Figure 4: Number of fragments in ERASURE(6, n) needed to achieve certain availability targets, as estimated by Glacier's single failure size model and our distribution-based model of G(0.0012, 0.4, 0.98). We also plot the actual achieved availability under the trace.

above that failure pattern prediction based on failure history may not be effective. By identifying the root causes of correlated failures, however, it becomes possible to predict their patterns in some cases [21]. For example, in a heterogeneous environment, nodes running the same OS may fail together due to viruses exploiting a common vulnerability.

On the other hand, first, not all root causes result in correlated failures with predictable patterns. For example, DDoS attacks result in correlated failures patterns that are inherently unpredictable. Second, it can be challenging to identify/predict root causes for all major correlated failures based on intuition. Human intuition does not reason well about close-to-zero probabilities [16]. As a result, it can be difficult to enumerate all top root causes. For example, should power failure be considered when targeting 5 nines availability? How about a certain vulnerability in a certain software? It is always possible to miss certain root causes that lead to major failures. Finally, it can also be challenging to identify/predict root causes for all major correlated failures, because the frequency of individual root causes can be extremely low. In some cases, a root cause may only demonstrate itself once (e.g., patches are usually installed after vulnerabilities are exploited). For the above reasons, we believe that while root cause analysis will help to make correlated failures more predictable, systems may never reach a point where we can predict all major correlated failures.

6 Is Simple Modeling of Failure Sizes Adequate?

A key challenge in system design is to obtain the "right" set of parameters that will be neither overly-pessimistic nor overly-optimistic in achieving given design goals. Because of the complexity in availability estimation introduced by failure correlation, system designers sometimes make simplifying assumptions on correlated failure sizes in order to make the problem more amenable. For example,

Glacier [19] considers only the (single) maximum failure size, aiming to achieve a given availability target despite the correlated failure of up to a fraction f of all the nodes. This simplifying assumption enables the system to use a closed-form formula [19] to estimate availability and then calculate the needed m and n values in ERASURE(m, n).

Our Finding. Figure 4 quantifies the effectiveness of this approach. The figure plots the number of fragments needed to achieve given availability targets under Glacier's model (with f=0.65 and f=0.45) and under WS_trace. Glacier does not explicitly explain how f can be chosen in various systems. However, we can expect that f should be a constant under the same deployment context (e.g., for WS_trace).

A critical point to observe in Figure 4 is that for the real trace, the curve is not a straight line (we will explore the shape of this curve later). Because the curves from Glacier's estimation are roughly straight lines, they always significantly depart from the curve under the real trace, regardless of how we tune f. For example, when f = 0.45, Glacier over-estimates system availability when n is large: Glacier would use ERASURE(6, 32) for an availability target of 7 nines, while in fact, ERASURE(6, 32) achieves only slightly above 5 nines availability. Under the same f, Glacier also under-estimates the availability of ERASURE(6, 10) by roughly 2 nines. If f is chosen so conservatively (e.g., f = 0.65) that Glacier never over-estimates, then the under-estimation becomes even more significant. As a result, Glacier would suggest ERASURE(6, 31) to achieve 3 nines availability while in reality, we only need to use ERASURE(6, 9). This would unnecessarily increase the bandwidth needed to create, update, and repair the object, as well as the storage overhead, by over 240%.

While it is obvious that simplifying assumptions (such as assuming a single failure size) will always introduce inaccuracy, our above results show that the inaccuracy can be dramatic (instead of negligible) in practical settings.

The Subtlety. The reason behind the above mismatch between Glacier's estimation and the actual availability under WS_trace is that in real systems, failure sizes may cover a large range. In the limit, failure events of any size may occur; the only difference is their likelihood. System availability is determined by the combined effects of failures with different sizes. Such effects cannot be summarized as the effects of a series of failures of the same size (even with scaling factors).

To avoid the overly-pessimistic or overly-optimistic configurations resulting from Glacier's method, a system must consider a distribution of failure sizes. IRISSTORE uses the bi-exponential model for this purpose. Figure 4 also shows the number of fragments needed for a given availability target as estimated by our simulator driven by the bi-exponential model. The estimation based on our model

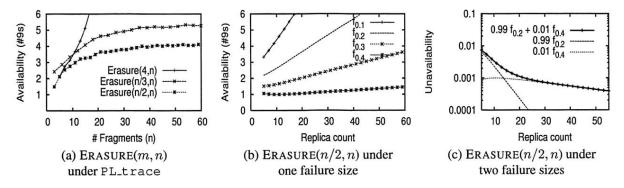


Figure 5: Availability of ERASURE(m,n) under different correlated failures. In (b) and (c), the legend f_x denotes correlated failures of x fraction of all nodes in the system, and qf_x denotes f_x happening with probability q.

matches the curve from WS_trace quite well. It is also important to note that the difference between Glacier's estimation and our estimation is purely from the difference between single failure size and a distribution of failure sizes. It is not because Glacier uses a formula while we use simulation. In fact, we have also performed simulations using only a single failure size, and the results are similar to those obtained using Glacier's formula.

The Design Principle. Assuming a single failure size can result in dramatic rather than negligible inaccuracies in practice. Thus correlated failures should be modeled via a distribution.

7 Are Additional Fragments Always Effective in Improving Availability?

Under independent failures, distributing the data always helps to improve availability, and any target availability can be achieved by using additional (smaller) fragments, without increasing the storage overhead. For this reason, system designers sometimes fix the ratio between nand m (so that the storage overhead, n/m, is fixed), and then simply increase n and m simultaneously to achieve For instance, under independent a target availability. failures, ERASURE(16, 32) gives better availability than ERASURE(12, 24), which, in turn, gives better availability than ERASURE(8, 16). Several existing systems, including OceanStore and CFS, use such ERASURE(n/2, n)schemes. Given independent failures, it can even be proved [29] that increasing the number of fragments (while keeping storage constant) exponentially decreases unavailability. In Section 9, we will show that a read/write replication system using majority voting [35] for consistency has the same availability as ERASURE(n/2, n). Thus, our discussion in this section also applies to these read/write replication systems (where, in this case, the storage overhead does increase with n).

Our Finding. We observe that distributing fragments across more and more machines (i.e., increasing n) may not be effective under correlated failures, even if we double or triple n. Figure 5(a) plots the availability of ERASURE(4,n), ERASURE(n/3,n) and ERASURE(n/2,n) under PL_trace. The graphs for WS_trace are similar (see [29]). We do not use RON_trace because it contains only 30 nodes. The three schemes clearly incur different storage overhead and achieve different availability. We do not intend to compare their availability under a given n value. Instead, we focus on the scaling trend (i.e., the availability improvement) under the three schemes when we increase n.

Both ERASURE(n/3,n) and ERASURE(n/2,n) suffer from a strong diminishing return effect. For example, increasing n from 20 to 60 in ERASURE(n/2,n) provides less than a half nine's improvement. On the other hand, ERASURE(4,n) does not suffer from such an effect. By tuning the parameters in the correlation model and using model-driven simulation, we further observe that the diminishing returns become more prominent under stronger correlation levels as well as under larger m values [29].

The above diminishing return shows that correlated failures prevent a system from effectively improving availability by distributed data across more and more machines. In read/write systems using majority voting, the problem is even worse: the same diminishing return effect occurs despite the significant increases in storage overhead.

The Subtlety. It may appear that diminishing return is an obvious consequence of failure correlation. However, somewhat surprisingly, we find that diminishing return does not directly result from the presence of failure correlation.

Figure 5(b) plots the availability of ERASURE(n/2, n) under synthetic correlated failures. In these experiments, the system has 277 nodes total (to match the system size in PL_trace), and each synthetic correlated failure crashes a random set of nodes in the system.⁴ The synthetic corre-

⁴Crashing *random* sets of nodes is our explicit choice for the experimental setup. The reason is exactly the same as in our bi-exponential failure correlation model (see Section 4).

lated failures all have the same size. For example, to obtain the curve labeled " $f_{0.2}$ ", we inject correlated failure events according to Poisson arrival, and each event crashes 20% of the 277 nodes in the system. Using single-size correlated failures matches the correlation model used in Glacier's configuration analysis.

Interestingly, the curves in Figure 5(b) are roughly straight lines and exhibit no diminishing returns. If we increase the severeness (i.e., size) of the correlated failures, the only consequence is smaller slopes of the lines, instead of diminishing returns. This means that diminishing return does not directly result from correlation. Next we explain that diminishing return actually results from the combined effects of failures with different sizes (Figure 5(c)).

Here we will discuss unavailability instead of availability, so that we can "sum up" unavailability values. In the real world, correlated failures are likely to have a wide range of sizes, as observed from our failure traces. Furthermore, larger failures tend to be rarer than smaller failures. As a rough approximation, system unavailability can be viewed as the summation of the unavailability caused by failures with different sizes. In Figure 5(c), we consider two different failure sizes $(0.2 \times 277 \text{ and } 0.4 \times 277)$, where the failures with the larger size occur with smaller probability. It is interesting to see that when we add up the unavailability caused by the two kinds of failures, the resulting curve is not a straight line. In fact, if we consider nines of availability (which flips the curve over), the combined curve shows exactly the diminishing return effect.

Given the above insight, we believe that diminishing returns are likely to generalize beyond our three traces. As long as failures have different sizes and as long as larger failures are rarer than smaller failures, there will likely be diminishing returns. The only way to lessen diminishing returns is to use a smaller m in ERASURE(m,n) systems. This will make the slope of the line for $0.01f_{0.4}$ in Figure 5(c) more negative, effectively making the combined curve straighter.

The Design Principle. System designers should be aware that correlated failures result in strong diminishing return effects in ERASURE(m,n) systems unless m is kept small. For popular systems such as ERASURE(n/2,n), even doubling or tripling n provides very limited benefits after a certain point.

8 Do Superior Designs under Independent Failures Remain Superior under Correlated Failures?

Traditionally, system designs are evaluated and compared under the assumption of independent failures. A positive

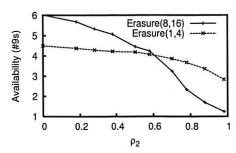


Figure 6: Effects of the correlation model $G(0.0012, \frac{2}{5}\rho_2, \rho_2)$ on two systems. $\rho_2 = 0$ indicates independent failures, while $\rho_2 = 0.98$ indicates roughly the correlation level observed in WS_trace.

answer to this question would provide support for this approach.

Our Finding. We find that correlated failures hurt some designs much more than others. Such non-equal effects are demonstrated via the example in Figure 6. Here, we plot the unavailability of ERASURE(1, 4) and ERASURE(8, 16) under different failure correlation levels, by tuning the parameters in our correlation model in model-driven simulation. These experiments use the model $G(0.0012, \frac{2}{5}\rho_2, \rho_2)$ (a generalization of the model for WS_trace to cover a range of ρ_2), and a universe of 130 nodes, with MTTF = 10 days and MTTR = 1 day. ERASURE(1, 4) achieves around 1.5 fewer nines than ERASURE(8, 16) when failures are independent (i.e., $\rho_2 = 0$). On the other hand, under the correlation level found in WS_trace ($\rho_2 = 0.98$), ERASURE(1, 4) achieves 2 more nines of availability than ERASURE(8, 16).

The Subtlety. The cause of this (perhaps counter-intuitive) result is the diminishing return effect described earlier. As the correlation level increases, ERASURE(8,16), with its larger m, suffers from the diminishing return effect to a much greater degree than ERASURE(1,4). In general, because diminishing return effects are stronger for systems with larger m, correlated failures hurt systems with large m more than those with small m.

The Design Principle. A superior design under independent failures may not be superior under correlated failures. In particular, correlation hurts systems with larger m more than those with smaller m. Thus designs should be explicitly evaluated and compared under correlated failures.

9 Read/Write Systems

Thus far, our discussion has focused on read-only ERASURE(m,n) systems. Interestingly, our results extend quite naturally to read/write systems that use quorum techniques or voting techniques to maintain data consistency.

⁵Note that the y-axis is on log-scale, and that is why the summation of two straight lines does not result in a third straight line.

 $^{^6\}mathrm{The}$ same conclusion also holds if we directly use WS_trace to drive the simulation.

Quorum systems (or voting systems) [7, 35] are standard techniques for maintaining consistency for read/write data. We first focus on the case of pure replication. Here, a user reading or writing a data object needs to access a *quorum* of replicas. For example, the majority quorum system (or majority voting) [35] requires that the user accesses a majority of the replicas. This ensures that any reader intersects in at least one replica with the latest writer, so that the reader sees the latest update. If a majority of the replicas is not available, then the data object is unavailable to the user. From an availability perspective, this is exactly the same as a read-only ERASURE(n/2, n) system. Among all quorum systems that always guarantee consistency, we will consider only majority voting because its availability is provably optimal [7].

Recently, Yu [40] proposed signed quorum systems (SQS), which uses quorum sizes smaller than n/2, at the cost of a tunably small probability of reading stale data. Smaller quorum sizes help to improve system availability because fewer replicas need to be available for a data object to be available. If we do not consider regeneration (i.e, creating new replicas to compensate for lost replicas), then the availability of such SQS-based systems would be exactly the same as that of a read-only ERASURE(m,n) system, where m is the quorum size. Regeneration makes the problem slightly more complex, but our design principles still apply [29].

Finally, quorum systems can also be used over erasure-coded data [18]. Despite the complexity of these protocols, they all have simple threshold-based requirements on the number of available fragments. As a result, their availability can also be readily captured by properly adjusting the m in our ERASURE(m,n) results.

10 IRISSTORE: A Highly-Available Read/Write Storage Layer

Applying the design principles in the previous sections, we have built IRISSTORE, a decentralized read/write storage layer that is highly-available despite correlated failures. IRISSTORE does not try to avoid correlated failures by predicting correlation patterns. Rather, it tolerates them by choosing the right set of parameters. IRISSTORE determines system parameters using a model with a failure size distribution rather than a single failure size. Finally, IRIS-STORE explicitly quantifies and compares configurations via online simulation using our correlation model. IRIS-STORE allows both replication and erasure coding for data redundancy, and also implements both majority quorum systems and SQS for data consistency. The original design of SQS [40] appropriately bounds the probability of inconsistency (e.g., probability of stale reads) when nodes MTTF and MTTR are relatively large compared to the inter-arrival time between writes and reads. If MTTF and MTTR are

small compared to the write/read inter-arrival time, the inconsistency may be amplified [5]. To avoid such undesirable amplification, IRISSTORE also incorporates a simple data refresh technique [28].

We use IRISSTORE as the read-write storage layer of IRISLOG⁷, a wide-area network monitoring system deployed on over 450+ PlanetLab nodes. IRISLOG with IRISSTORE has been available for public use for over 8 months.

At a high level, IRISSTORE consists of two major components. The first component replicates or encodes objects, and processes user accesses. The second component is responsible for regeneration, and automatically creates new fragments when existing ones fail. These two subsystems have designs similar to Om [41]. For lack of space, we omit the details (see [26]) and focus on two aspects that are unique to IRISSTORE.

10.1 Achieving Target Availability

Applications configure IRISSTORE by specifying an availability target, a bi-exponential failure correlation model, as well as a cost function. The correlation model can be specified by saying that the deployment context is "PlanetLablike," "WebServer-like," or "RON-like." In these cases, IRISSTORE will use one of the three built-in failure correlation models from Section 4. We also intend to add more built-in failure correlation models in the future. To provide more flexibility, IRISSTORE also allows the application to directly specify the three tunable parameters in the bi-exponential distribution. It is our long term goal to extend IRISSTORE to monitor failures in the deployment, and automatically adjust the correlation model if the initial specification is not accurate.

The cost function is an application-defined function that specifies the overall cost resulting from performance overhead and inconsistency (for read/write data). The function takes three inputs, m, n, and i (for inconsistency), and returns a cost value that the system intends to minimize given that the availability target is satisfied. For example, a cost function may bound the storage overhead (i.e., n/m) by returning a high cost if n/m exceeds certain threshold. Similarly, the application can use the cost function to ensure that not too many nodes need to be contacted to retrieve the data (i.e., bounding m). We choose to leave the cost function to be completely application-specific because the requirements from different applications can be dramatically different.

With the cost function and the correlation model, IRIS-STORE uses online simulation to determine the best values for m, n, and quorum sizes. It does so by exhaustively searching the parameter space (with some practical caps on n and m), and picking the configuration that minimizes

⁷http://www.intel-iris.net/irislog

the cost function while still achieving the availability target. The amount of inconsistency (i) is predicted [39, 40] based on the quorum size. Finally, this best configuration is used to instantiate the system. Our simulator takes around 7 seconds for each configuration (i.e., each pair of m and n values) on a single 2.6GHz Pentium 4; thus IRISSTORE can perform a brute-force exhaustive search for 20,000 configurations (i.e., a cap of 200 for both n and m, and $m \le n$) in about one and a half days. Many optimizations are possible to further prune the search space. For example, if ERASURE(16, 32) does not reach the availability target then neither does ERASURE(17, 32). This exhaustive search is performed offline; its overhead does not affect system performance.

10.2 Optimizations for Regeneration

When regenerating read/write data, IRISSTORE (like RAMBO [25] and Om [41]) uses the Paxos distributed consensus protocol [23] to ensure consistency. Using such consistent regeneration techniques in the wide-area, however, poses two practical challenges that are not addressed by RAMBO and Om:⁸

Flooding problem. After a large correlated failure, one instance of the Paxos protocol needs to be executed for each of the objects that have lost any fragments. For example, our IRISLOG deployment on the PlanetLab has 3530 objects storing PlanetLab sensor data, and each object has 7 replicas. A failure of 42 out of 206 PlanetLab nodes (on 3/28/2004) would flood the system with 2,478 instances of Paxos. Due to the message complexity of Paxos, this creates excessive overhead and stalls the entire system.

Positive feedback problem. Determining whether a distant node has failed is often error-prone due to unpredictable communication delays and losses. Regeneration activity after a correlated failure increases the inaccuracy of failure detection due to the increased load placed on the network and nodes. Unfortunately, inaccurate failure detections trigger more regeneration which, in turn, results in larger inaccuracy. Our experience shows that this positive feedback loop can easily crash the entire system.

To address the above two problems, IRISSTORE implements two optimizations:

Opportunistic Paxos-merging. In IRISSTORE, if the system needs to invoke Paxos for multiple objects whose fragments *happen* to reside on the same set of nodes, the regeneration module merges all these Paxos invocations into a single one. This approach is particularly effective with IRISSTORE's load balancing algorithm [27], which tends to place the fragments for two objects either on the same set of nodes or on completely disjoint sets of nodes. Compared to explicitly aggregating objects into clusters, this

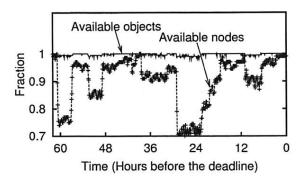


Figure 7: Availability of IRISSTORE in the wild.

approach avoids the need to maintain consistent split and merge operations on clusters.

Paxos admission-control. To avoid the positive feedback problem, we use a simple admission control mechanism on each node to control its CPU and network overhead, and to avoid excessive false failure detections. Specifically, each node, before initiating a Paxos instance, samples (by piggybacking on the periodic ping messages) the number of ongoing Paxos instances on the relevant nodes. Paxos is initiated only if the average and the maximum number of ongoing Paxos instances are below some thresholds (2 and 5, respectively, in our current deployment). Otherwise, the node queues the Paxos instance and backs off for some random time before trying again. Immediately before a queued Paxos is started, IRISSTORE rechecks whether the regeneration is still necessary (i.e., whether the object is still missing a fragment). This further improves failure detection accuracy, and also avoids regeneration when the failed nodes have already recovered.

10.3 Availability of IRISSTORE in the Wild

In this section and the next, we report our availability and performance experience with IRISSTORE (as part of IRISLOG) on the PlanetLab over an 8-month period. IRISLOG uses IRISSTORE to maintain 3530 objects for different sensor data collected from PlanetLab nodes. As a background testing and logging mechanism, we continuously issued one query to IRISLOG every five minutes. Each query tries to access all the objects stored in IRISSTORE. We set a target availability of 99.9% for our deployment, and IRISSTORE chooses, accordingly, a replica count of 7 and a quorum size of 2.

Figure 7 plots the behavior of our deployment under stress over a 62-hour period, 2 days before the SOSP'05 deadline (i.e., 03/23/2005). The PlanetLab tends to suffer the most failures and instabilities right before major conference deadlines. The figure shows both the fraction of available nodes in the PlanetLab and the fraction of available (i.e., with accessible quorums) objects in IRISSTORE.

⁸RAMBO has never been deployed over the wide-area, while Om has never been evaluated under a large number of simultaneous node failures.

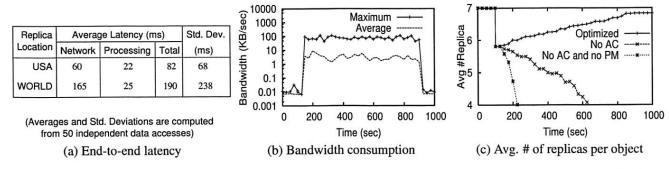


Figure 8: IRISSTORE performance. (In (c), "AC" means Paxos admission-control and "PM" means Paxos-merging.)

At the beginning of the period, IRISSTORE was running on around 200 PlanetLab nodes. Then, the PlanetLab experienced a number of large correlated failures, which correspond to the sharp drops in the "available nodes" curve (the lower curve). On the other hand, the fluctuation of the "available objects" curve (the upper curve) is consistently small, and is always above 98% even during such an unusual stress period. This means that our real world IRISSTORE deployment tolerated these large correlated failures well. In fact, the overall availability achieved by IRISSTORE during the 8-month deployment was 99.927%, demonstrating the ability of IRISSTORE to achieve a preconfigured target availability.

10.4 Basic Performance of IRISSTORE

Access latency. Figure 8(a) shows the measured latency from our lab to access a single object in our IRISLOG deployment. The object is replicated on random PlanetLab nodes. We study two different scenarios based on whether the replicas are within the US or all over the world. Network latency contributes to most of the end-to-end latency, and also to the large standard deviations.

Bandwidth usage. Our next experiment studies the amount of bandwidth consumed by regeneration, including the Paxos protocol. To do this, we consider the PlanetLab failure event on 3/28/2004 that caused 42 out of the 206 live nodes to crash in a short period of time (an event found by analyzing PL_trace). We replay this event by deploying IRISLOG on 206 PlanetLab nodes and then killing the IRISLOG process on 42 nodes.

Figure 8(b) plots the bandwidth used during regeneration. The failures are injected at time 100. We observe that on average, each node only consumes about 3KB/sec bandwidth, out of which 2.8KB/sec is used to perform necessary regeneration, 0.27KB/sec for unnecessary regeneration (caused by false failure detection), and 0.0083KB/sec for failure detection (pinging). The worst-case peak for an individual node is roughly 100KB/sec, which is sustainable even with home DSL links.

Regeneration time. Finally, we study the amount of time

needed for regeneration, demonstrating the importance of our Paxos-merging and Paxos admission-control optimizations. We consider the same failure event (i.e., killing 42 out of 206 nodes) as above.

Figure 8(c) plots the average number of replicas per object and shows how IRISSTORE gradually regenerates failed replicas after the failure at time 100. In particular, the failure affects 2478 objects. Without Paxos admission-control or Paxos-merging, the regeneration process does not converge.

Paxos-merging reduces the total number of Paxos invocations from 2478 to 233, while admission-control helps the regeneration process to converge. Note that the average number of replicas does not reach 7 even at the end of the experiment because some objects lose a majority of replicas and cannot regenerate. A single regeneration takes 21.6sec on average which is dominated by the time for data transfer (14.3sec) and Paxos (7.3sec). The time for data transfer is determined by the amount of data and can be much larger. The convergence time of around 12 minutes is largely determined by the admission-control parameters and may be tuned to be faster. However, regeneration is typically not a time-sensitive task, because IRIS-STORE only needs to finish regeneration before the next failure hits. Our simulation study based on PL_trace shows that 12-minute regeneration time achieves almost identical availability as, say, 5-minute regeneration time. Thus we believe IRISSTORE's regeneration mechanism is adequately efficient.

11 Conclusion

Despite the wide awareness of correlated failures in the research community, the properties of such failures and their impact on system behavior has been poorly understood. In this paper, we made some critical steps toward helping system developers understand the impact of realistic, correlated failures on their systems. In addition, we provided a set of design principles that systems should use to tolerate such failures, and showed that some existing approaches are less effective than one might expect. We presented the

design and implementation of a distributed read/write storage layer, IRISSTORE, that uses these design principles to meet availability targets even under real-world correlated failures.

Acknowledgments. We thank the anonymous reviewers and our shepherd John Byers for many helpful comments on the paper. We also thank David Anderson for giving us access to RON_trace, Jay Lorch for his comments on an earlier draft of the paper, Jeremy Stribling for explaining several aspects of PL_trace, Hakim Weatherspoon for providing the code for failure pattern prediction, Jay Wylie for providing WS_trace, and Praveen Yalagandula for helping us to process PL_trace. This research was supported in part by NSF Grant No. CNS-0435382 and funding from Intel Research.

References

- [1] Akamai Knocked Out, Major Websites Offline. http://techdirt.com/articles/20040524/0923243.shtml.
- [2] Emulab network emulation testbed home. http://www.emulab.net.
- [3] PlanetLab All Pair Pings. http://www.pdos.lcs.mit.edu/~strib/pl_app/.
- [4] The MIT RON trace. David G. Andersen, Private communication.
- [5] AIYER, A., ALVISI, L., AND BAZZI, R. On the Availability of Non-strict Quorum Systems. In DISC (2005).
- [6] BAKKALOGLU, M., WYLIE, J., WANG, C., AND GANGER, G. On Correlated Failures in Survivable Storage Systems. Tech. Rep. CMU-CS-02-129, Carnegie Mellon University, 2002.
- [7] BARBARA, D., AND GARCIA-MOLINA, H. The Reliability of Voting Mechanisms. IEEE Transactions on Computers 36, 10 (1987).
- [8] BHAGWAN, R., TATI, K., CHENG, Y., SAVAGE, S., AND VOELKER, G. M. TotalRecall: Systems Support for Automated Availability Management. In USENIX NSDI (2004).
- [9] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In ACM SIGMETRICS (2000).
- [10] CATES, J. Robust and Efficient Data Management for a Distributed Hash Table. Masters Thesis, Massachusetts Institute of Technology, 2003.
- [11] CHUN, B., AND VAHDAT, A. Workload and Failure Characterization on a Large-Scale Federated Testbed. Tech. Rep. IRB-TR-03-040, Intel Research Berkeley, 2003.
- [12] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-Diagonal Parity for Double Disk Failure Correction. In *USENIX FAST* (2004).
- [13] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In ACM SOSP (2001).
- [14] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for Low Latency and High Throughput. In USENIX NSDI (2004).
- [15] DOUCEUR, J. R., AND WATTENHOFER, R. P. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In DISC (2001).
- [16] FRIED, B. H. Ex Ante/Ex Post. Journal of Contemporary Legal Issues 13, 1 (2003).
- [17] GIBBONS, P. B., KARP, B., KE, Y., NATH, S., AND SESHAN, S. IrisNet: An Architecture for a Worldwide Sensor Web. *IEEE Pervasive Computing* 2, 4 (2003).

- [18] GOODSON, G., WYLIE, J., GANGER, G., AND REITER, M. Efficient Byzantine-tolerant Erasure-coded Storage. In *IEEE DSN* (2004).
- [19] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In USENIX NSDI (2005).
- [20] HARCHOL-BALTER, M., AND DOWNEY, A. B. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. ACM Transactions on Computer Systems 15, 3 (1997).
- [21] JUNQUEIRA, F., BHAGWAN, R., HEVIA, A., MARZULLO, K., AND VOELKER, G. Surviving Internet Catastrophes. In USENIX Annual Technical Conference (2005).
- [22] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATH-ERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In ACM ASPLOS (2000).
- [23] LAMPORT, L. The Part-Time Parliament. ACM Transactions on Computer Systems 16, 2 (1998).
- [24] LELAND, W. E., TAQQ, M. S., WILLINGER, W., AND WILSON, D. V. On the Self-similar Nature of Ethernet Traffic. In ACM SIG-COMM (1993).
- [25] LYNCH, N., AND SHVARTSMAN, A. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In DISC (2002).
- [26] NATH, S. Exploiting Redundancy for Robust Sensing. PhD thesis, Carnegie Mellon University, 2005. Tech. Rep. CMU-CS-05-166.
- [27] NATH, S., GIBBONS, P. B., AND SESHAN, S. Adaptive Data Placement for Wide-Area Sensing Services. In USENIX FAST (2005).
- [28] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Tolerating Correlated Failures in Wide-Area Monitoring Services. Tech. Rep. IRP-TR-04-09, Intel Research Pittsburgh, 2004.
- [29] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in Tolerating Correlated Failures in Wide-Area Storage Systems. Tech. Rep. IRP-TR-05-52, Intel Research Pittsburgh, 2005. Also available at http://www.cs.cmu.edu/~yhf/correlated.pdf.
- [30] NURMI, D., BREVIK, J., AND WOLSKI, R. Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments. In *Euro-Par* (2005).
- [31] PLANK, J. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. Software – Practice & Experience 27, 9 (1997).
- [32] ROWSTRON, A., AND DRUSCHEL, P. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In ACM SOSP (2001).
- [33] SHI, J., AND MALIK, J. Normalized Cuts and Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence 22, 8 (2000).
- [34] TANG, D., AND IYER, R. K. Analysis and Modeling of Correlated Failures in Multicomputer Systems. *IEEE Transactions on Comput*ers 41, 5 (1992).
- [35] THOMAS, R. H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. ACM Transactions on Database Systems 4, 2 (1979).
- [36] WEATHERSPOON, H., CHUN, B.-G., SO, C. W., AND KUBIA-TOWICZ, J. Long-Term Data Maintenance: A Quantitative Approach. Tech. Rep. UCB/CSD-05-1404, University of California, Berkeley, 2005.
- [37] WEATHERSPOON, H., MOSCOVITZ, T., AND KUBIATOWICZ, J. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. In *IEEE RPPDS* (2002).
- [38] YALAGANDULA, P., NATH, S., Yu, H., GIBBONS, P. B., AND SESHAN, S. Beyond Availability: Towards a Deeper Understanding of Machine Failure Characteristics in Large Distributed Systems. In USENIX WORLDS (2004).
- [39] YU, H. Overcoming the Majority Barrier in Large-Scale Systems. In DISC (2003).
- [40] YU, H. Signed Quorum Systems. In ACM PODC (2004).
- [41] YU, H., AND VAHDAT, A. Consistent and Automatic Replica Regeneration. ACM Transactions on Storage 1, 1 (2005).

Open Versus Closed: A Cautionary Tale

Bianca Schroeder Carnegie Mellon University Pittsburgh, PA 15213 bianca@cs.cmu.edu Adam Wierman

Carnegie Mellon University

Pittsburgh, PA 15213

acw@cs.cmu.edu

Mor Harchol-Balter Carnegie Mellon University Pittsburgh, PA 15213 harchol@cs.cmu.edu

Abstract

Workload generators may be classified as based on a closed system model, where new job arrivals are only triggered by job completions (followed by think time), or an open system model, where new jobs arrive independently of job completions. In general, system designers pay little attention to whether a workload generator is closed or open.

Using a combination of implementation and simulation experiments, we illustrate that there is a vast difference in behavior between open and closed models in real-world settings. We synthesize these differences into eight simple guiding principles, which serve three purposes. First, the principles specify how scheduling policies are impacted by closed and open models, and explain the differences in user level performance. Second, the principles motivate the use of partly open system models, whose behavior we show to lie between that of closed and open models. Finally, the principles provide guidelines to system designers for determining which system model is most appropriate for a given workload.

1 Introduction

Every systems researcher is well aware of the importance of setting up one's experiment so that the system being modeled is "accurately represented." Representing a system accurately involves many things, including accurately representing the bottleneck resource behavior, the scheduling of requests at that bottleneck, and workload parameters such as the distribution of service request demands, popularity distributions, locality distributions, and correlations between requests. However, one factor that researchers typically pay little attention to is whether the job arrivals obey a closed or an open system model. In a closed system model, new job arrivals are only triggered by job completions (followed by think time), as in Figure 1(a). By contrast in an open system model, new jobs arrive independently of job completions, as in Figure 1(b).

Table 1 surveys the system models in a variety of web related workload generators used by systems researchers today. The table is by no means complete; however it illustrates the wide range of workload generators and benchmarks available. Most of these generators/benchmarks assume a closed system model, although a reasonable fraction assume an open one. For many of these workload generators, it was quite difficult to figure out which system model was being assumed the builders often do not seem to view this as an important factor worth mentioning in the documentation. Thus the "choice" of a system model (closed versus open) is often not really a researcher's choice, but rather is dictated by the availability of the workload generator. Even when a user makes a conscious choice to use a closed model, it is not always clear how to parameterize the closed system (e.g. how to set the think time and the multiprogramming level - MPL) and what effect these parameters will have.

In this paper, we show that closed and open system models yield significantly different results, even when both models are run with the same load and service demands. Not only is the measured response time different under the two system models, but the two systems respond fundamentally differently to varying parameters and to resource allocation (scheduling) policies.

We obtain our results primarily via real-world implementations. Although the very simplest models of open and closed systems can be compared analytically, analysis alone is insufficient to capture the effect of many of the complexities of modern computer systems, especially size based scheduling and realistic job size distributions. Real-world implementations are also needed to capture the magnitude of the differences between closed and open systems in practice. The case studies we consider are described in Section 4. These include web servers receiving static HTTP requests in both a LAN and a WAN setting; the back-end database in e-commerce applications; and an auctioning web site. In performing these

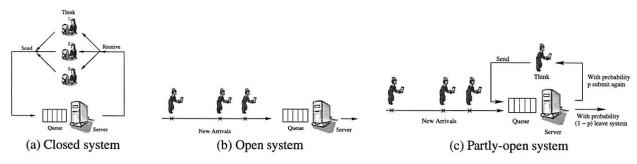


Figure 1: Illustrations of the closed, open, and partly-open system models.

case studies, we needed to develop a flexible suite of workload generators, simulators, and trace analysis tools that can be used under closed, open, and other system models. The details of this suite are also provided in Section 4.

Our simulation and implementation experiments lead us to identify *eight principles*, summarizing the observed differences between open and closed system models, many of which are not obvious. These principles may be categorized by their area of impact.

The first set of principles (see Section 5.1) describe the difference in mean response time under open and closed system models and how various parameters affect these differences. We find, for example, that for a fixed load, the mean response time for an open system model can exceed that for a closed system model by an order of magnitude or more. Even under a high MPL, the closed system model still behaves "closed" with respect to mean response time, and there is still a significant difference between mean response times in closed and open systems even for an MPL of 1000. With respect to service demands (job sizes), while their variability has a huge impact on response times in open systems, it has much less of an effect in closed models. The impact of these principles is that a system designer needs to beware of taking results that were discovered under one system model (e.g. closed model) and applying them to a second system model (e.g. open model).

The second set of principles (see Section 5.2) deal with the *impact of scheduling* on improving system performance. Scheduling is a common mechanism for improving mean response time without purchasing additional resources. While Processor-Sharing scheduling (PS) and First-Come-First-Served (FCFS) are most commonly used in computer systems, many system designs give preference to short jobs (requests with small service demands), applying policies like Non-Preemptive-Shortest-Job-First (SJF) or Preemptive-Shortest-Job-First (PSJF) to disk scheduling [51] and web server scheduling [19, 33, 15]. When system designers seek to evaluate a new scheduling policy, they often try it out using a workload generator and simulation test-bed. Our work will show

that, again, one must be very careful that one is correctly modeling the application as closed or open, since the impact of scheduling turns out to be very different under open and closed models. For example, our principles show that favoring short jobs is highly effective in improving mean response time in open systems, while this is far less true under a closed system model. We find that closed system models only benefit from scheduling under a narrow range of parameters, when load is moderate and the MPL is very high. The message for system designers is that understanding whether the workload is better modeled with an open or closed system is essential in determining the effectiveness of scheduling.

The third set of principles (see Section 6) deal with partly-open systems. We observe that while workload generators and benchmarks typically assume either an open system model or a closed system model, neither of these is entirely realistic. Many applications are best represented using an "in-between" system model, which we call the partly-open model. Our principles specify those parameter settings for which the partly-open model behaves more like a closed model or more like an open model with respect to response time. We also find that, counter to intuition, parameters like think time have almost no impact on the performance of a partlyopen model. The principles describing the behavior of the partly-open system model are important because realworld applications often fit best into partly-open models, and the performance of these models is not well understood. In particular, the effect of system parameters and scheduling on performance in the partly open system points which our principles address - are not known. Our results motivate the importance of designing versatile workload generators that are able to support open, closed, and partly open system models. We create such versatile workload generators for several common systems, including web servers and database systems, and use these throughout our studies.

The third set of principles also provides system designers with guidelines for how to choose a system model when they are forced to pick a workload generator that is either purely closed or purely open, as are almost all

Type of benchmark	Name	System model
Model-based web workload generator	Surge [10], WaspClient [31], Geist [22], WebStone [47], WebBench [49], MS Web Capacity Analysis Tool [27]	Closed
	SPECWeb96 [43], WAGON [23]	Open
Playback mechanisms for HTTP request streams	MS Web Application Stress Tool [28], Webjamma [2], Hammerhead [39], Deluge [38], Siege [17]	Closed
	httperf [30], Sclient [9]	Open
Proxy server benchmarks	Wisconsin Proxy Benchmark [5], Web Polygraph [35], Inktomi Climate Lab [18]	Closed
Database benchmark for e-commerce workloads	TPC-W [46]	Closed
Auction web site benchmark	RUBiS[7]	Closed
Online bulletin board benchmark	RUBBoS[7]	Closed
Database benchmark for online transaction processing (OLTP)	TPC-C [45]	Closed
Model-based packet level web traffic generators	IPB (Internet Protocol Benchmark) [24], GenSyn [20] WebTraf [16], trafgen [14]	Closed
	NS traffic generator [52]	Open
Mail server benchmark	SPECmail2001 [42]	Open
Java Client/Server benchmark	SPECJ2EE [41]	Open
Web authentication and authorization	AuthMark [29]	Closed
Network file servers	NetBench [48]	Closed
	SFS97_R1 (3.0) [40]	Open
Streaming media service	MediSyn [44]	Open

Table 1: A summary table of the system models underlying standard web related workload generators.

workload generators (see Section 7). We consider ten different workloads and use our principles to determine for each workload which system model is most appropriate for that workload: closed, open, or partly-open. To the best of our knowledge, no such guide exists for systems researchers.

2 Closed, open, and partly-open systems

In this section, we define how requests are generated under closed, open, and partly-open system models.

Figure 1(a) depicts a closed system configuration. In a closed system model, it is assumed that there is some fixed number of users, who use the system forever. This number of users is typically called the multiprogramming level (MPL) and denoted by N. Each of these N users repeats these 2 steps, indefinitely: (a) submit a job, (b) receive the response and then "think" for some amount of time. In a closed system, a new request is only triggered by the completion of a previous request. At all times there are some number of users, N_{think} , who are thinking, and some number of users N_{system} , who are either running or queued to run in the system, where $N_{think} + N_{system} = N$. The response time, T, in a closed system is defined to be the time from when a request is submitted until it is received. In the case where the system is a single server (e.g. a web server), the server load, denoted by ρ , is defined as the fraction of time that the server is busy, and is the product of the mean throughput X and the mean service demand (processing requirement) E[S].

Figure 1(b) depicts an **open system** configuration. In an open system model there is a stream of arriving users with average arrival rate λ . Each user is assumed to submit one job to the system, wait to receive the response, and then leave. The number of users queued or running at the system at any time may range from zero to infinity. The differentiating feature of an open system is that a request completion does not trigger a new request: a new request is only triggered by a new user arrival. As before, response time, T, is defined as the time from when a request is submitted until it is completed. The server load is defined as the fraction of time that the server is busy. Here load, ρ , is the product of the mean arrival rate of requests, λ , and the mean service demand E[S].

Neither the open system model nor the closed system model is entirely realistic. Consider for example a web site. On the one hand, a user is apt to make more than one request to a web site, and the user will typically wait for the output of the first request before making the next. In these ways a closed system model makes sense. On the other hand, the number of users at the site varies over time; there is no sense of a fixed number of users N. The point is that users visit to the web site, behave as if they are in a closed system for a short while, and then leave the system.

Motivated by the example of a web site, we study a more realistic alternative to the open and closed system configurations: the partly-open system shown in Figure 1(c). Under the partly-open model, users arrive according to some outside arrival process as in an open system. However, every time a user completes a request at the system, with probability p the user stays and makes a followup request (possibly after some think time), and with probability 1 - p the user simply leaves the system. Thus the expected number of requests that a user makes to the system in a visit is Geometrically distributed with mean 1/(1-p). We refer to the collection of requests a user makes during a visit to the system as a session and we define the *length* of a session to be the number of requests in the session/visit. The server load is the fraction of time that the server is busy equaling the product of the average outside arrival rate λ , the mean number of requests per visit E[R], and the mean service demand E[S]. For a given load, when p is small, the partly-open model is more similar to an open model. For large p, the partly-open model resembles a closed model.

3 Comparison methodology

In this section we discuss the relevant parameters and metrics for both the open and the closed system models and discuss how we set parameters in order to compare open and closed system models.

Throughout the paper we choose the service demand distribution to be the same for the open and the closed system. In the case studies the service demand distribution is either taken from a trace or determined by the benchmark used in the experiments. In the model-based simulation experiments later in the paper, we use hyperexponential service demands, in order to capture the highly variable service distributions in web applications. Throughout, we measure the variability in the service demand distribution using the square coefficient of variation, C^2 . The think time in the closed system, Z, follows an exponential distribution, and the arrival process in the open system is either a Poisson arrival process with average rate λ , or is provided by traces. The results for all simulations and experiments are presented in terms of mean response times and the system load ρ . While we do not explicitly report numbers for another important metric, mean throughput, the interested reader can directly infer those numbers by interpreting load as a simple scaling of throughput. In an open system, the mean throughput is simply equal to $\lambda = \rho/E[S]$, which is the same as throughput in a closed system.

In order to fairly compare the open and closed systems, we will hold the system load ρ for the two systems equal, and study the effect of open versus closed system models on mean response time. The load in the open system is specified by λ , since $\rho = \lambda E[S]$. Fixing the load of a closed system is more complex, since the load is affected by many parameters including the MPL, the think time, the service demand variability, and the scheduling policy. The fact that system load is influenced by many more system parameters in a closed system than in an open system is a surprising difference between the two systems. Throughout, we will achieve a desired system load by adjusting the think time of the closed system (see Figure 7(a)), while holding all other parameters fixed.

The scheduling policies we study in this work span the range of behaviors of policies that are used in computer systems today.

- **FCFS** (First-Come-First-Served) Jobs are processed in the same order as they arrive.
- **PS** (Processor-Sharing) The server is shared evenly among all jobs in the system.
- **PESJF** (Preemptive-Expected-Shortest-Job-First) The job with the smallest expected duration (size) is given preemptive priority.
- **SRPT** (Shortest-Remaining-Processing-Time-First): At every moment the request with the smallest remaining processing requirement is given priority.
- **PELJF** (Preemptive-Expected-Longest-Job-First) The job with the longest expected size is given preemptive priority. PELJF is an example of a policy that performs badly and is included to understand the full range of possible response times.

4 Real-world case studies

In this section, we compare the behavior of four different applications under closed, open, and partly open system models. The applications include (a) a web server delivering static content in a LAN environment, (b) the database back-end at an e-commerce web site, (c) the application server at an auctioning web site, and (d) a web server delivering static content in a WAN environment. These applications vary in many respects, including the bottleneck resource, the workload properties (e.g. job size variability), network effects, and the types of scheduling policies considered. We study applications (a), (b), and (d) through full implementation in a real testbed, while our study of application (c) relies on trace-based simulation.

As part of the case studies, we develop a set of workload generators, simulators, and trace analysis tools that facilitate experimentation with all three system models: open, closed, and partly-open. For implementationbased case studies we extend the existing workload generator (which is based on only one system model) to

¹Note that we choose a Poisson arrival process (i.e. exponential inter-arrival times) and exponential think times in order to allow the open and closed systems to be as parallel as possible. This setting underestimates the differences between the systems when more bursty arrival processes are used.

enable all three system models. For the case studies based on trace-driven simulation, we implement a versatile simulator that models open, closed, and partly-open systems and takes traces as input. We also develop tools for analyzing web traces (in Common Logfile Format or Squid log format) to extract the data needed to parameterize workload generators and simulators.

Sections 4.1 – 4.4 provide the details of the case studies. The main results are shown in Figures 2 and 4. For each case study we first explain the tools developed for experimenting in open, closed, and partly-open models. We then then describe the relevant scheduling policies and their implementation, and finally discuss the results. The discussions at the end of the case studies are meant only to highlight the key points; we will discuss the differences between open, closed, and partly-open systems and the impact of these differences in much more detail in Sections 5 and 6.

4.1 Static web content

Our first case study is an Apache web server running on Linux and serving static content, i.e. requests of the form "Get me a file," in a LAN environment. Our experimental setup involves six machines connected by a 10/100 Ethernet switch. Each machine has an Intel Pentium III 700 MHz processor and 256 MB RAM, and runs Linux. One of the machines is designated as the server and runs Apache. The others generate web requests based on a web trace.

Workload generation: In this case study we generate static web workloads based on a trace. Below we first describe our workload generator which generates web requests following an open, closed, or partly-open model. We then describe the tool for analyzing web traces that produces input files needed by the workload generator. Finally we briefly describe the actual trace that we are using in our work.

Our workload generator is built on top of the Sclient[9] workload generator. The Sclient workload generator uses a simple open system model, whereby a new request for file y is made exactly every x msec. Sclient is designed as a single process that manages all connections using the select system call. After each call to select, Sclient checks whether the current x msec interval has passed and if so initiates a new request. We generalize Sclient in several ways.

For the open system, we change Sclient to make requests based on arrival times and filenames specified in an input file. The entries in the input file are of the form $\langle t_i, f_i \rangle$, where t_i is a time and f_i is a file name.

For the closed system, the input file only specifies the names of the files to be requested. To implement closed system arrivals in Sclient, we have Sclient maintain a list with the times when the next requests are to be made.

Entries to the list are generated during runtime as follows: Whenever a request completes, an exponentially distributed think time Z is added to the current time t_{curr} and the result $Z+t_{curr}$ is inserted into the list of arrival times.

In the case of the partly-open system, each entry in the input file now defines a session, rather than an individual request. An entry in the input file takes the form $< t_i, f_{i_1}, \ldots, f_{i_n} >$ where t_i specifies the arrival time of the session and $< f_{i_1}, \ldots, f_{i_n} >$ is the list of files to be requested during the session. As before, a list with arrival times is maintained according to which requests are made. The list is initialized with the session arrival times t_i from the input file. To generate the arrivals within a session, we use the same method as described for the closed system above: after request $f_{i_{j-1}}$ completes we arrange the arrival of request f_{i_j} by adding an entry containing the arrival time $Z + t_{curr}$ to the list, where t_{curr} is the current time and Z is an exponentially distributed think time.

All the input files for the workload generator are created based on a web trace. We modify the Webalizer tool [12] to parse a web trace and then extract the information needed to create the input files for the open, closed, and partly-open system experiments. In the case of the open system, we simply output the arrival times together with the names of the requested files. In the case of the closed system, we only extract the sequence of file names. Creating the input file for the partly-open system is slightly more involved since it requires identifying the sessions in a trace. A common approach for identifying sessions (and the one taken by Webalizer) is to group all successive requests by the same client (i.e. same IP address) into one session, unless the time between two requests exceeds some timeout threshold in which case a new session is started. In our experiments, we use the timeout parameter to specify the desired average session length.

The trace we use consists of one day from the 1998 World Soccer Cup, obtained from the Internet Traffic Archive [21]. Virtually all requests in this trace are *static*.

Number	Mean	Variability	Min	Max
of Req.	size	(C^2)	size	size
$4.5 \cdot 10^{6}$	5KB	96	41 bytes	2MB

Scheduling: Standard scheduling of static requests in a web server is best modeled by processor sharing (PS). However, recent research suggests favoring requests for small files can improve mean response times at web servers [19]. In this section we therefore consider both PS and SRPT policies.

We have modified the Linux kernel and the Apache Web server to implement SRPT scheduling at the server. For static HTTP requests, the network (access link out of the server) is typically the bottleneck resource. Thus, our solution schedules the bandwidth on this access link by

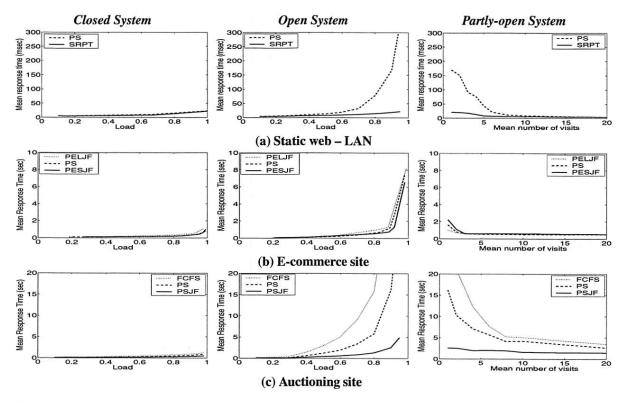


Figure 2: Results for real-world case studies. Each row shows the results for a real-world workload and each column shows the results for one of the system models. In all experiments with the closed system model the MPL is 50. The partly-open system is run at fixed load 0.9.

controlling the order in which the server's socket buffers are drained. Traditionally, the socket buffers are drained in Round-Robin fashion (similar to PS); we instead give priority to sockets corresponding to connections where the remaining data to be transferred is small. Figure 3 shows the flow of data in Linux after our modifications. There are multiple priority queues and queue *i* may only

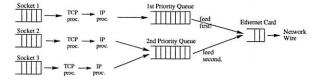


Figure 3: Flow of data in Linux with SRPT-like scheduling (only 2 priority levels shown).

drain if queues 0 to i-1 are empty. The implementation is enabled by building the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queuing, and the Prio Pseudoscheduler and by using the tc[6] user space tool. We also modify Apache to use setsockopt calls to update the priority of the socket as the remaining size of the transfer decreases. For details on our implementation see [19].

Synopsis of results: Figure 2(a) shows results from the

the static web implementation under closed, open, and partly open workloads in a LAN environment. Upon first glance, it is immediately clear that the closed system response times are vastly different from the open response times. In fact, the response times in the two systems are orders of magnitude different under PS given a common system load. Furthermore, SRPT provides little improvement in the closed system, while providing dramatic improvement in the open system.

The third column of Figure 2(a) shows the results for the partly-open system. Notice that when the mean number of requests is small, the partly-open system behaves very much like the open system. However, as the mean number of requests grows, the partly-open system behaves more like a closed system. Thus, the impact of scheduling (e.g. SRPT over PS) is highly dependent on the number of requests in the partly-open system.

4.2 E-commerce site

Our second case study considers the database back-end server of an e-commerce site, e.g. an online bookstore. We use a PostgreSQL[32] database server running on a 2.4-GHz Pentium 4 with 3GB RAM, running Linux 2.4, with a buffer pool of 2GB. The machine is equipped with two 120GB IDE drives, one used for the database log and the other for the data. The workload is generated by

four client machines having similar specifications to the database server connected via a network switch.

Workload generation: The workload for the e-commerce case study is based on the TPC-W [46] benchmark, which aims to model an online bookstore such as Amazon.com. We build on the TPC-W kit provided by the Pharm project [13]. The kit models a closed system (in accordance with TPC-W guidelines) by creating one process for each client in the closed system.

We extend the kit to also support an open system with Poisson arrivals, and a partly-open system. We do so by creating a master process that signals a client whenever it is time to make a new request in the open system or to start a new session in the partly-open system. The master process repeats the following steps in a loop: it sleeps for an exponential interarrival time, signals a client, and draws the next inter-arrival time. The clients block waiting for a signal from the master process. In the case of the open system, after receiving the signal, the clients make one request before they go back to blocking for the next signal. In the case of the partly-open system, after receiving a signal, the clients generate a session by executing the following steps in a loop: (1) make one request; (2) flip a coin to decide whether to begin blocking for a signal from the master process or to generate an exponential think time and sleep for that time.

TPC-W consists of 16 different transaction types including the "ShoppingCart" transaction, the "Payment" transaction, and others. Statistics of our configuration are as shown:

Database	Mean	Variability (C^2)	Min	Max
size	size		size	size
3GB	101 ms	4	2 ms	5s

Scheduling: The bottleneck resource in our setup is the CPU, as observed in [25]. The default scheduling policy is therefore best described as PS, in accordance with Linux CPU scheduling. Note that in this application, exact service demands are not known, so SRPT cannot be implemented. Thus, we experiment with PESJF and PELJF policies where the expected service demand of a transaction is based on its type. The "Bestseller" transaction, which makes up 10% of all requests, has on average the largest service demand. Thus, we study 2-priority PESJF and PELJF policies where the "Bestseller" transactions are "expected to be long" and all other transactions are "expected to be short."

To implement the priorities needed for achieving PESJF and PELJF, we modify our PostgreSQL server as follows. We use the sched_setscheduler() system call to set the scheduling class of a PostgreSQL process working on a high priority transaction to "SCHED_RR," which marks a process as a Linux real-time process. We leave the scheduling class of a low pri-

ority process at the standard "SCHED_OTHER." Realtime processing in Linux always has absolute, preemptive priority over standard processes.

Synopsis of results: Figure 2(b) shows results from the e-commerce implementation described above. Again, the difference in response times between the open and closed systems is immediately apparent – the response times of the two systems differ by orders of magnitude. Interestingly, because the variability of the service demands is much smaller in this workload than in the static web workload, the impact of scheduling in the open system is much smaller. This also can be observed in the plot for the partly open system: even when the number of requests is small, there is little difference between the response times of the different scheduling policies.

4.3 Auctioning web site

Our third case study investigates an auctioning web site. This case study uses simulation based on a trace from one of the top-ten U.S. online auction sites.

Workload generation: For simulation-based case studies we implement a simulator that supports open, closed, and partly-open arrival processes which are either created based on a trace or are generated from probability distributions. For a trace-based arrival process the simulator expects the same input files as the workload generator described in Section 4.1. If no trace for the arrival process is available the simulator alternatively offers (1) open system arrivals following a Poisson process; (2) closed system arrivals with exponential think times; (3) partly-open arrivals with session arrivals following a Poisson process and think times within the sessions being exponentially distributed. The service demands can either be specified through a trace or one of several probability distributions, including hyper-exponential distributions and more general distributions.

For our case study involving an auctioning web site we use the simulator and a trace containing the service demands obtained from one of the top ten online auctioning sites in the US. No data on the request arrival process is available. The characteristics of the service demands recorded in the trace are summarized below:

Number	Mean	Variability	Min	Max
of jobs	size	(C^2)	size	size
300000	0.09s	9.19	0.01s	50s

Scheduling: The policy used in a web site serving dynamic content, such as an auctioning web site, is best modeled by PS. To study the effect of scheduling in this environment we additionally simulate FCFS and PSJF.

Synopsis of results: Figure 2(c) shows results from the auctioning trace-based case study described above. The plots here illustrate the same properties that we observed in the case of the static web implementation. In fact, the

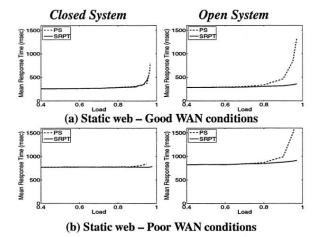


Figure 4: Effect of WAN conditions in the static web case study. The top row shows results for good WAN conditions (average RTT=50ms, loss rate=1%) and the bottom row shows results for poor WAN conditions (average RTT=100ms, loss rate=4%). In both cases the closed system has an MPL of 200. Note that, due to network effects, the closed system cannot achieve a load of 1, even when think time is zero. Under the settings we consider here, the max achievable load is ≈ 0.98 .

difference between the open and closed response times is extreme, especially under FCFS. As a result, there is more than a factor of ten improvement of PSJF over FCFS (for $\rho > 0.7$), whereas there is little difference in the closed system.

This effect can also be observed in the partly-open system, where for a small number of requests per session the response times are comparable to those in the open system and for a large number of requests per session the response times are comparable to those in the closed system. The actual convergence rate depends on the variability of the service demands (C^2) . In particular, the ecommerce case study (low C^2) converges quickly, while the static web and auctioning case studies (higher C^2) converge more slowly.

4.4 Study of WAN effects

To study the effect of network conditions, we return to the case of static web requests (Section 4.1), but this time we include the emulation of network losses and delays in the experiments.

Workload generation: The setup and workload generation is identical to the case study of static web requests (Section 4.1), except that we add functionality for emulating WAN effects as follows. We implement a separate module for the Linux kernel that can drop or delay incoming and outgoing TCP packets (similarly to Dummynet [34] for FreeBSD). More precisely, we change the ip_rcv() and the ip_output() functions in the Linux TCP-IP stack to intercept in- and out-going packets to create losses and delays. In order to delay packets,

we use the add_timer() facility to schedule the transmission of delayed packets. We recompile the kernel with HZ=1000 to get a finer-grained millisecond timer resolution. In order to drop packets, we use an independent, uniform random loss model which can be configured to a specified probability, as in Dummynet.

Synopsis of results: Figure 4 compares the response times of the closed and the open systems under (a) relatively good WAN conditions (50ms RTT and 1% loss rate) and under (b) poor WAN conditions (100ms RTT and 4% loss rate). Note that results for the partly-open system are not shown due to space constraints; however the results parallel what is shown in the closed and open systems.

We find that under WAN conditions the differences between the open and closed systems are smaller (proportionally) than in a LAN (Figure 2 (a)), however, they are still significant for high server loads (load > 0.8). The reason that the differences are smaller in WAN conditions is that response times include network overheads (network delays and losses) in addition to delays at the server. These overheads affect the response times in the open and closed systems in the same way, causing the proportional differences between open and closed systems to shrink. For similar reasons, scheduling has less of an effect when WAN effects are strong, even in the case of an open system. SRPT improves significantly over PS only for high loads, and even then the improvement is smaller than in a LAN.

5 Open versus closed systems

We have just seen the dramatic impact of the system model in real-world case studies. We will now develop principles that help explain both the differences between the open and closed system and the impact of these differences with respect to scheduling. In addition to the case studies that we have already discussed, we will also use model-based simulations in order to provide more control over parameters, such as job size variability, that are fixed in the case studies.

5.1 FCFS

Our study of the simple case of FCFS scheduling will illustrate three principles that we will exploit when studying more complex policies.

Principle (i): For a given load, mean response times are significantly lower in closed systems than in open systems.

Principle (i) is maybe the most noticeable performance issue differentiating open and closed systems in our case studies (Figure 2). We bring further attention to this principle in Figure 5 due to its importance for the vast literature on capacity planning, which typically relies

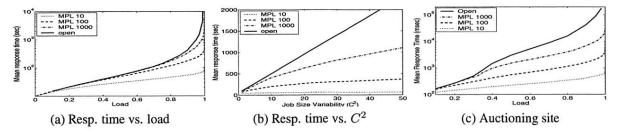


Figure 5: Open versus closed under FCFS. Model and trace-based simulation results showing mean response time as a function of load and service demand variability under FCFS scheduling. (a) and (b) use model based simulation, while (c) uses trace-based simulation. In all cases, the solid line represents an open system and the dashed lines represent closed systems with different MPLs. The load is adjusted via the think time in the closed system, and via the arrival rate in the open system. In the model-based simulations, E[S] = 10. In (a) we fix $C^2 = 8$ and in (b) we fix $\rho = 0.9$.

on closed models, and hence may underestimate the resources needed when an open model is more appropriate.

For fixed high loads, the response time under the closed system is *orders of magnitude* lower than those for the open system. While Schatte [36, 37] has proven that, under FCFS, the open system will always serve as an upper bound for the response time of the closed system, the magnitude of the difference in practical settings has not previously been studied. Intuitively, this difference in mean response time between open and closed systems is a consequence of the fixed MPL, N, in closed systems, which limits the queue length seen in closed systems to N even under very high load. By contrast, no such limit exists for an open system.

Principle (ii): As the MPL grows, closed systems become open, but convergence is slow for practical purposes.

Principle (ii) is illustrated by Figure 5. We see that as the MPL, N, increases from 10 to 100 to 1000, the curves for the closed system approach the curves for the open system. Schatte [36, 37] proves formally that as N grows to infinity, a closed FCFS queue converges to an open M/GI/1/FCFS queue. What is interesting however, is how slowly this convergence takes place. When the service demand has high variability (C^2), a closed system with an MPL of 1000 still has much lower response times then the corresponding open system. Even when the job service demands are lightly variable, an MPL of 500 is required for the closed system to achieve response times comparable to the corresponding open system. Further, the differences are more dramatic in the case-study results than in the model-based simulations.

This principle impacts the choice of whether an open or closed system model is appropriate. One might think that an open system is a reasonable approximation for a closed system with a high MPL; however, though this can be true in some cases, the closed and open system models may still behave significantly differently if the service demands are highly variable.

Principle (iii): While variability has a large effect in

open systems, the effect is much smaller in closed systems

This principle is difficult to see in the case-study figures (Figure 2) since each trace has a fixed variability. However, it can be observed by comparing the magnitude of disparity between the e-commerce site results (low variability) and the others (high variability).

Using simulations, we can study this effect directly. Figure 5(b) compares open and closed systems under a fixed load $\rho=0.9$, as a function of the service demand variability C^2 . For an open system, we see that C^2 directly affects mean response time. This is to be expected since high C^2 , under FCFS service, results in short jobs being stuck behind long jobs, increasing mean response time. By contrast, for the closed system with MPL 10, C^2 has comparatively little effect on mean response time. This is counterintuitive, but can be explained by observing that for lower MPL there are fewer short jobs stuck behind long jobs in a closed system, since the number of jobs in the system (N_{system}) is bounded. As MPL is increased, C^2 can have more of an effect, since N_{system} can be higher.

It is important to point out that by holding the load constant in Figure 5(b), we are actually performing a conservative comparison of open and closed systems. If we didn't hold the load fixed as we changed C^2 , increasing C^2 would result in a slight drop in the load of the closed system as shown in Figure 7(b). This slight drop in load, would cause a drop in response times for closed systems, whereas there is no such effect in open systems.

5.2 The impact of scheduling

The value of scheduling in open systems is understood and cannot be overstated. In open systems, there are order of magnitude differences between the performance of scheduling policies because scheduling can prevent small jobs from queueing behind large jobs. In contrast, scheduling in closed systems is not well understood.

Principle (iv): While open systems benefit significantly

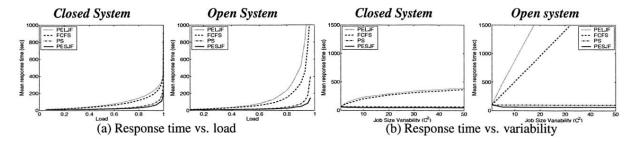


Figure 6: Model-based simulation results illustrating the different effects of scheduling in closed and open systems. In the closed system the MPL is 100, and in both systems the service demand distribution has mean 10. For the two figures in (a) C^2 was fixed at 8 and in the two figures in (b) the load was fixed at 0.9.

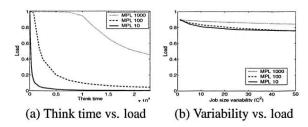


Figure 7: Model-based simulation results illustrating how the service demand variability, the MPL, and the think time can affect the system load in a closed system. These plots use FCFS scheduling, however results are parallel under other scheduling policies.

from scheduling with respect to response time, closed systems improve much less.

Principle (v): Scheduling only significantly improves response time in closed systems under very specific parameter settings: moderate load (think times) and high MPL.

Figure 2 illustrates the fundamentally different behavior of mean response time in the open and closed systems in realistic settings. In Figure 6, we further study this difference as a function of (a) load and (b) variability using simulations. Under the open system, as load increases, the disparity between the response times of the scheduling policies grows, eventually differing by orders of magnitude. In contrast, at both high and low loads in the closed system, the scheduling policies all perform similarly; only at moderate loads is there a significant difference between the policies – and even here the differences are only a factor of 2.5. Another interesting point is that, whereas for FCFS the mean response time of an open system bounded that in the corresponding closed system from above, this does not hold for other policies such as PESJF, where the open system can result in lower response times than the closed system.

We can build intuition for the limited effects of scheduling in closed systems by first considering a closed feedback loop with no think time. In such a system, surprisingly, the scheduling done at the queue is inconsequential – all work conserving scheduling policies per-

form equivalently. To see why, note that in a closed system Little's Law states that N=XE[T], where N is the constant MPL across policies. We will now explain why X is constant across all work conserving scheduling policies (when think time is 0), and hence it will follow that E[T] is also constant across scheduling policies. X is the long-run average rate of completions. Since a new job is only created when a job completes, over a long period of time, all work conserving scheduling policies will complete the same set of jobs plus or minus the initial set N. As time goes to infinity, the initial set N becomes unimportant; hence X is constant. This argument does not hold for open systems because for open systems Little's Law states that $E[N] = \lambda E[T]$, and E[N] is not constant across scheduling policies.

Under closed systems with think time, we now allow a varying number of jobs in the queue, and thus there is some difference between scheduling policies. However, as think time grows, load becomes small and so scheduling has less effect.

A very subtle effect, not yet mentioned, is that in a closed system the scheduling policy actually affects the throughput, and hence the load. "Good" policies, like PESJF, increase throughput, and hence load, slightly (less than 10%). Had we captured this effect (rather than holding the load fixed), the scheduling policies in the closed system would have appeared even closer, resulting in even starker differences between the closed and open systems.

The impact of Principles (iv) and (v) is clear. For closed systems, scheduling provides small improvement across all loads, but can only result in substantial improvement when load (think time) is moderate. In contrast, scheduling always provides substantial improvements for open systems.

Principle (vi): Scheduling can limit the effect of variability in both open and closed systems.

For both the open and closed systems, better scheduling (PS and PESJF) helps combat the effect of increasing variability, as seen in Figure 6. The improvement; how-

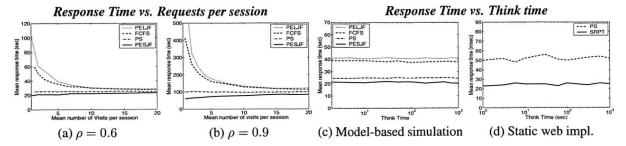


Figure 8: Model and implementation-based results for the partly-open system. (a) and (b) are model-based simulations showing mean response time as a function of the expected number of requests per session. (c) and (d) show the mean response time as a function of the think time, for a fixed load. In (a)-(c), E[S] = 10 and $C^2 = 8$. In (c) and (d), we fix $\rho = 0.6$ and $\rho = 0.75$, which yields and average of 4 requests per session.

ever, is less dramatic for closed systems due to Principle (iii) in Section 5.1, which tells us that variability has less of an effect on closed systems in general.

6 Partly-open systems

In this section, we discuss a partly-open model that (a) serves as a more realistic system model for many applications; and (b) helps illustrate when a "purely" open or closed system is a good approximation of user behavior. We focus on the effects of the mean number of requests per session and the think time because the other parameters, e.g. load and job size variability, have similar effects to those observed in Sections 5.1 and 5.2. Throughout the section, we fix the load of the partly-open system by adjusting the arrival rate, λ . Note that, in contrast to the closed model, adjusting the think time of the partly-open model has no impact on the load.

Principle (vii): A partly-open system behaves similarly to an open system when the expected number of requests per session is small (≤ 5 as a rule-of-thumb) and similarly to a closed system when the expected number of requests per session is large (≥ 10 as a rule-of-thumb).

Principle (vii) is illustrated clearly in the case study results shown in Figure 2 and in the simulation results shown in Figure 8(a). When the mean number of requests per session is 1 we have a significant separation between the response time under the scheduling policies, as in open systems. However, when the mean number of requests per session is large, we have comparatively little separation between the response times of the scheduling policies; as in closed systems. Figures 2 and 8(a) are just a few examples of the range of configurations we studied, and across a wide range of parameters, the point where the separation between the performance of scheduling policies becomes small is, as a rule-of-thumb, around 10 requests per session. Note however that this point can range anywhere between 5 and 20 requests per session as C^2 ranges from 4 to 49 respectively. We will demonstrate in Section 7 how to use this rule-of-thumb as a guideline for determining whether a purely open or purely closed workload generator is most suitable, or whether a partly-open generator is necessary.

Principle (viii): In a partly-open system, think time has little effect on mean response time.

Figure 8 illustrates Principle (viii). We find that the think time in the partly-open system does not affect the mean response time or load of the system under any of these policies. This observation holds across all partly-open systems we have investigated (regardless of the number of requests per session), including the case-studies described in Section 4.

Principle (viii) may seem surprising at first, but for PS and FCFS scheduling it can be shown formally under product-form workload assumptions. Intuitively, we can observe that changing the think time in the partly-open system has no effect on the load because the same amount of work must be processed. To change the load, we must adjust either the number of requests per session or the arrival rate. The only effect of think time is to add small correlations into the arrival stream.

7 Choosing a system model

The previous sections brought to light vast differences in system performance depending on whether the workload generator follows an open or closed system model. A direct consequence is that the accuracy of performance evaluation depends on accurately modeling the underlying system as either open, closed, or partly-open.

A safe way out would be to choose a partly-open system model, since it both matches the typical user behavior in many applications and generalizes the open and closed system models – depending on the parameters it can behave more like an open or more like a closed system. However, as Table 1 illustrates, available workload generators often support only either closed or open system models. This motivates a fundamental questions for workload modeling: "Given a particular workload, is a purely open or purely closed system model more accurate

	Type of site	Date	Total #Req.
1	Large corporate web site	Feb'01	1609799
2	CMU web server [3]	Nov'01	90570
3	Online department store	June'00	891366
4	Science institute (USGS[1])	Nov'02	107078
5	Online gaming site [50]	May'04	45778
6	Financial service provider	Aug'00	275786
7	Supercomputing web site [4]	May'04	82566
8	Kasparov-DeepBlue match	May'97	580068
9	Site seeing "slashdot effect"	Feb'99	194968
10	Soccer world cup [21]	Jul'98	4606052

Table 2: A summary table of the studied web traces.

for the workload? When is a partly-open system model necessary?"

In the remainder of this section we illustrate how our eight principles might be used to answer this question for various web workloads. Our basic method is as follows. For a given system we follow these steps:

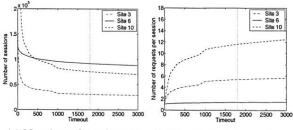
- 1. Collect traces from the system.
- Construct a partly-open model for the system, since the partly-open model is the most general and accurate. In particular, obtain the relevant parameters for the partly-open model.
- For the partly-open model, decide whether an open or a closed model is appropriate, or if the partlyopen model is necessary.

Table 2 summarizes the traces we collected as part of Step 1. Our trace collection spans many different types of sites, including busy commercial sites, sites of major sporting events, sites of research institutes, and an online gaming site.

We next model each site as a partly-open system. According to Principles (vii) and (viii) the most relevant parameter of a partly-open system model is the number of requests issued in a user session. Other parameters such as the think time between successive requests in a session are of lesser importance. Determining the average number of requests per user session for a web site requires identifying user sessions in the corresponding web trace. While there is no 100% accurate way to do this, we employ some common estimation techniques [8, 26].

First, each source IP address in a trace is taken to represent a different user. Second, session boundaries are determined by a period of inactivity by the user, i.e. a period of time during which no requests from the corresponding IP address are received. Typically, this is accomplished by ending a session whenever there is a period of inactivity larger than timeout threshold τ . In some cases, web sites themselves enforce such a threshold; however, more typically τ must be estimated.

We consider two different ways of estimating τ . The first one is to use a defacto standard value for τ , which



(a) Number of sessions vs Timeout length

(b) Number of requests vs Timeout length

Figure 9: Choosing a system model. Statistics for 3 representative web traces (sites 3, 6, and 10) illustrating (a) the number of user sessions as a function of the timeout threshold and (b) the expected number of requests per session as a function of the timeout threshold. The vertical line on each plot corresponds to a timeout of 1800s. From these plots we can conclude that an open model is appropriate for site 6, a closed model is appropriate for site 10, and neither an open or a closed is appropriate for site 3.

is 1800s (30 min) [26]. The second method is to estimate τ from the traces themselves by studying the derivative of how τ affects the total number of sessions in the trace. We illustrate this latter method for a few representative traces in Figure 9(a). Notice that as the threshold increases from 1-100s the number of sessions decreases quickly; whereas from 1000s on, the decrease is much smaller. Furthermore, Figure 9(b) shows that with respect to the number of requests, stabilization is also reached at $\tau > 1000$ s. Hence we adopt $\tau = 1800$ s in what follows.

The mean number of requests per session when $\tau = 1800s$ is summarized below for all traces:

Site	1	2	3	4	5
Requests per session	2.4	1.8	5.4	3.6	12.9
Site	6	7	8	9	10
Requests per session	1.4	6.0	2.4	1.2	11.6

The table indicates that the average number of requests for web sessions varies largely depending on the site, ranging from less than 2 requests per session to almost 13. Interestingly, even for similar types of web sites the number of requests can vary considerably. For example sites 8 and 10 are both web sites of sporting events (a chess tournament and a soccer tournament), but the number of requests per session is quite low (2.4) in one case, while quite high (11.6) in the other. Similarly, sites 2, 4, and 7 are all web sites of scientific institutes but the number of requests per sessions varies from 1.8 to 6.

Using the rule of thumb in principle (vii), we can conclude that neither the open nor the closed system model accurately represents all the sites. For sites 1, 2, 4, 6, 8, and 9 an open system model is accurate; whereas a closed system model is accurate for the sites 5 and 10.

Further, it is not clear whether an open or closed model is appropriate for sites 3 and 7.

Observe that the web trace of site 10 is the same dataset used to drive the static web case study in Figure 2. In Figure 2, we observed a large difference between the response times in an open and a closed system model. In this section we found that site 10 resembles more a closed system than an open system. Based on the results in Figure 2, it is important that one doesn't assume that the results for the closed model apply to the open model.

8 Prior Work

Work explicitly comparing open and closed system models is primarily limited to FCFS queues. Bondi and Whitt [11] study a general network of FCFS queues and conclude that the effect of service variability, though dominant in open systems, is almost inconsequential in closed systems (provided the MPL is not too large). We corroborate this principle and illustrate the magnitude of its impact in real-world systems. Schatte [36, 37] studies a single FCFS queue in a closed loop with think time. In this model, Schatte proves that, as the MPL grows to infinity, the closed system converges monotonically to an open system. This result provides a fundamental understanding of the effect of the MPL parameter; however the rate of this convergence, which is important when choosing between open and closed system models, is not understood. We evaluate the rate of convergence in realworld systems.

Though these theoretical results provide useful intuition about the differences between open and closed systems, theoretical results alone cannot evaluate the effects of factors such as trace driven job service demand distributions, correlations, implementation overheads, and size-based scheduling policies. Hence, simulation and implementation-based studies such as the current paper are needed.

9 Conclusion

This paper provides eight simple principles that function to explain the differences in behavior of closed, open, and partly-open systems and validates these principles via trace-based simulation and real-world implementation. The more intuitive of these principles point out that response times under closed systems are typically lower than in the corresponding open system with equal load, and that as MPL increases, closed systems approach open ones. Less obviously, our principles point out that: (a) the magnitude of the difference in response times between closed and open systems can be very large, even under moderate load; (b) the convergence of closed to open as MPL grows is slow, especially when service demand variability (C^2) is high; and (c) scheduling is far more beneficial in open systems than in closed ones. We

also compare the partly-open model with the open and closed models. We illustrate the strong effect of the number of requests per session and C^2 on the behavior of the partly-open model, and the surprisingly weak effect of think time.

These principles underscore the importance of choosing the appropriate system model. For example, in capacity planning for an open system, choosing a workload generator based on a closed model can greatly underestimate response times and underestimate the benefits of scheduling.

All of this is particularly relevant in the context of web applications, where the arrival process at a web site is best modeled by a partly-open system. Yet, most web workload generators are either strictly open or strictly closed. Our findings provide guidelines for choosing whether an open or closed model is the better approximation based on characteristics of the workload. A high number of simultaneous users (more than 1000) suggests an open model, but a high number of requests per session (more than 10) suggests a closed model. Both these cutoffs are affected by service demand variability: highly variable demands requires larger cutoffs. Contrary to popular belief, it turns out that think times are irrelevant to the choice of an open or closed model since they only affect the load. We also find that WAN conditions (losses and delays) in Web settings lessen the difference between closed and open models, although these differences are still noticeable.

Once it has been determined whether a closed or open model is a better approximation, that in turn provides a guideline for the effectiveness of scheduling. Understanding the appropriate system model is essential to understanding the impact of scheduling. Scheduling is most effective in open systems, but can have moderate impact in closed systems when both the load is moderate (roughly 0.7-0.85) and C^2 is high.

In conclusion, while much emphasis has been placed in research on accurately representing workload parameters such as service demand distribution, think time, locality, etc, we have illustrated that similar attention needs to be placed on accurately representing the system itself as either closed, open, or partly-open. We have taken a first step toward this end by providing guidelines for choosing a system model and by creating tools and workload generators versatile enough to support all three system models. We hope that this work will encourage others to design workload generators that allow flexibility in the underlying system model.

10 Acknowledgments

We would like to thank Arun Iyengar, Erich Nahum, Paul Dantzig, Luis von Ahn, and Chad Vizino for providing access to the logs we used in Section 4.3 and Section 7.

This work was supported by an IBM PhD fellowship, NSF grants CCR-0133077, CCR-0311383, and CCR-0313148, and by IBM via TTC grant 2005-2006.

References

- [1] The U.S. Geological Survey. http://www.usgs.gov.
- [2] Webjamma world wide web traffic analysis tools. http://research.cs.vt.edu/chitra/webjamma.html.
- [3] Carnegie Mellon University, School of Computer Science. http://www.cs.cmu.edu/,2005.
- [4] The Pittsburgh Supercomputing Center (PSC). http://www.psc.edu/, 2005.
- [5] J. Almeida and P. Cao. Wisconsin proxy benchmark 1.0. http://www.cs.wisc.edu/cao/wpb1.0.html, 1998.
- [6] W. Almesberger. Linux network traffic control implementation overview. http://diffserv.sourceforge.net, 1999.
- [7] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In Workshop on Workload Characterization, 2002.
- [8] M. Arlitt. Characterizing web user sessions. SIGMETRICS Perform. Eval. Rev., 28(2):50–63, 2000.
- [9] G. Banga and P. Druschel. Measuring the capacity of a web server under realistic loads. World Wide Web, 2(1-2):69–83, 1999.
- [10] P. Barford and M. Crovella. The surge traffic generator: Generating representative web workloads for network and server performance evaluation. In *In Proc. of the ACM SIGMETRICS*, 1998.
- [11] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perf. Eval.*, 6:219–234, 1986.
- [12] Bradford L. Barrett. The Webalizer log file analysis program. http://www.mrunix.net/webalizer,2005.
- [13] T. Cain, M. Martin, T. Heil, E. Weglarz, and T. Bezenek. Java TPC-W implementation. http://www.ece.wisc.edu/~pharm/tpcw.shtml, 2000.
- [14] R. Chinchilla, J. Hoag, D. Koonce, H. Kruse, S. Ostermann, and Y. Wang. The trafgen traffic generator. In *Proc. of Int. Conf. on Telecommunication Sys.*, Mod. and Anal. (ICTSM10), 2002.
- [15] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In USENIX Symposium on Internet Technologies and Systems, October 1999.
- [16] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proc. ACM Sigcomm*, 1999.
- [17] J. Fulmer. Siege. http://joedog.org/siege.
- [18] S. Gigandet, A. Sudarsanam, and A. Aggarwal. The inktomi climate lab: an integrated environment for analyzing and simulating customer network traffic. In *Proc. ACM SIGCOMM Workshop on Int. Meas.*, pages 183–187, 2001.
- [19] M. Harchol-Balter, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. ACM Transactions on Computer Systems, 21(2), May 2003.
- [20] P. E. Heegaard. Gensyn generator of synthetic internet traffic. http://www.item.ntnu.no/~poulh/GenSyn-/gensyn.html.
- [21] ITA. The Internet traffic archives. Available at http://town.hall.org/Archives/pub/ITA/, 2002.
- [22] K. Kant, V. Tewari, and R. Iyer. GEIST: Generator of ecommerce and internet server traffic. In Proc. of Int. Symposium on Performance Analysis of Systems and Software, 2001.
- [23] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Performance Evaluation*, 46(2-3):77–100, 2001.

- [24] B. A. Mah, P. E. Sholander, L. Martinez, and L. Tolendino. Ipb: An internet protocol benchmark using simulated traffic. In MAS-COTS 1998, pages 77–84, 1998.
- [25] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Int. Conf. on Data Engineering*, 2004.
- [26] D. Menasce and V. Almeida. Scaling for E-Buisness: technologies, models, performance, and capacity planning. Prentice Hall, 2000
- [27] Microsoft IIS 6.0 Resource Kit Tools. Microsoft Web Capacity Analysis Tool (WCAT) version 5.2.
- [28] Microsoft TechNet. Ms web application stress tool (WAST).
- [29] Mindcraft. The AuthMark benchmark. http://www.-mindcraft.com/authmark/.
- [30] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3), 1998.
- [31] E. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on WWW server performance. In *Proc of* ACM SIGMETRICS, pages 257–267, 2001.
- [32] PostgreSQL. http://www.postgresql.org.
- [33] M. Rakwat and A. Kshemkayani. SWIFT: Scheduling in web servers for fast response time. In Symp. on Net. Comp. and App., April 2003.
- [34] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. ACM Comp. Comm. Review, 27(1), 1997.
- [35] A. Rousskov and D. Wessels. High performance benchmarking with web polygraph. Software - Practice and Experience, 1:1–10, 2003.
- [36] P. Schatte. On conditional busy periods in n/M/GI/1 queues. Math. Operationsforsh. u. Statist. ser. Optimization, 14, 1983.
- [37] P. Schatte. The M/GI/1 queue as limit of closed queueing systems. Math. Operationsforsh. u. Statist. ser. Optimization, 15:161-165, 1984.
- [38] sourceforge.net. Deluge a web site stress test tool. http://deluge.sourceforge.net/.
- [39] sourceforge.net. Hammerhead 2 web testing tool. http://hammerhead.sourceforge.net/.
- [40] Standard Performance Evaluation Corporation. SFS97_R1 (3.0). http://www.specbench.org/osg/web99/.
- [41] Standard Performance Evaluation Corporation. SPECJ2EE. http://www.specbench.org/osg/web99/.
- [42] Standard Performance Evaluation Corporation. SPECmail2001. http://www.specbench.org/osg/web99/.
- [43] Standard Performance Evaluation Corporation. SPECweb99. http://www.specbench.org/osg/web99/.
- [44] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat. Medisyn: A synthetic streaming media service workload generator. In Proceedings of 13th NOSSDAV, 2003.
- [45] Transaction Processing Performance Council. TPC benchmark C. Number Revision 5.1.0, December 2002.
- [46] Transaction Processing Performance Council. TPC benchmark W (web commerce). Number Revision 1.8, February 2002.
- [47] G. Trent and M. Sake. WebStone: The first generation in HTTP server benchmarking. http://www.mindcraft.com/webstone/paper.html.
- [48] VeriTest. Netbench 7.0.3. http://www.etestinglabs.com/benchmarks/netbench/.
- [49] VeriTest. Webbench 5.0. http://www.etestinglabs.com/benchmarks/webbench/.
- [50] L. von Ahn and L. Dabbish. Labeling images with a computer game. In CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 319–326, 2004.
- [51] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proc. of Sigmetrics*, 1994.
- [52] M. Yuksel, B. Sikdar, K. S. Vastola, and B. Szymanski. Work-load generation for ns simulations of wide area networks and the internet. In *Proc. of Comm. Net. and Dist. Sys. Mod. and Sim.*, 2000.

An Architecture for Internet Data Transfer

Niraj Tolia[†], Michael Kaminsky[‡], David G. Andersen[†], and Swapnil Patil[†]

†Carnegie Mellon University and [‡]Intel Research Pittsburgh

Abstract

This paper presents the design and implementation of DOT, a flexible architecture for data transfer. This architecture separates content negotiation from the data transfer itself. Applications determine *what* data they need to send and then use a new *transfer service* to send it. This transfer service acts as a common interface between applications and the lower-level network layers, facilitating innovation both above and below. The transfer service frees developers from re-inventing transfer mechanisms in each new application. New transfer mechanisms, in turn, can be easily deployed without modifying existing applications.

We discuss the benefits that arise from separating data transfer into a service and the challenges this service must overcome. The paper then examines the implementation of DOT and its plugin framework for creating new data transfer mechanisms. A set of microbenchmarks shows that the DOT prototype performs well, and that the overhead it imposes is unnoticeable in the wide-area. End-to-end experiments using more complex configurations demonstrate DOT's ability to implement effective, new data delivery mechanisms underneath existing services. Finally, we evaluate a production mail server modified to use DOT using trace data gathered from a live email server. Converting the mail server required only 184 lines-of-code changes to the server, and the resulting system reduces the bandwidth needed to send email by up to 20%.

1 Introduction

Bulk data transfers represent more than 70% of Internet traffic [3]. As a result, many efforts have examined ways to improve the efficiency and speed of these transfers, but these efforts face a significant deployment barrier: Most applications do not distinguish between their control logic and their data transfer logic. For example, HTTP and SMTP both interleave their control commands (e.g., the HTTP header, SMTP's "mail from:", etc.) and their data transfers over the same TCP connection. Therefore, new innovations in bulk data transfer must be reimplemented for each application. Not surprisingly, the rate of adoption of innovative transfer mechanisms, particularly in existing systems, is slow.

Data transfer applications typically perform two different functions. The first is *content negotiation*, which is very application-specific. For example, a Web download involves transmitting the name of the object, negotiating the language for the document, establishing a common format for images, and storing and sending cookies. The second function is *data transfer*, in which the actual data bits are exchanged. The process of data transfer is generally independent of the application, but applications and protocols almost always bundle these functions together.

Historically, data transfers have been tightly linked with content negotiation for several reasons. The first is likely expediency: TCP and the socket API provide a mechanism that is "good enough" for application developers who wish to focus on the other, innovative parts of their programs. The second reason is the challenge of naming. In order to transfer a data object, an application must be able to name it. The different ways that applications define their namespaces and map names to objects is one of the key differences between many protocols. For example, FTP and HTTP both define object naming conventions, and may provide different names for the same objects. Other protocols such as SMTP only name their objects implicitly during the data transfer.

As a concrete example of the cost of this coupling, consider the steps necessary to use BitTorrent [8] to accelerate the delivery of email attachments to mailing lists. Such an upgrade would require changes to each sender and receiver's SMTP servers, and modifications to the SMTP protocol itself. These changes, however, would only benefit email. To use the same techniques to speed Web downloads and reduce the load at Web servers would again require modification of both the HTTP protocol and servers.

We propose cleanly separating data transfer from applications. Applications still perform content negotiation using application-specific protocols, but they use a *transfer service* to perform bulk point-to-point data transfers. The applications pass the data object that they want to send to the transfer service. The transfer service is then responsible for ensuring that this object reaches the receiver. The simplest transfer service might accomplish this by sending the data via a TCP connection to the receiver. A more complex transfer service could implement the above BitTorrent data transfer techniques, making them available to SMTP, HTTP, and other applications.

Separating data transfer from the application provides

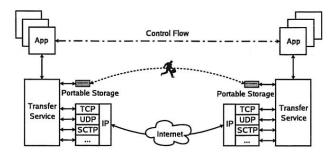


Figure 1: DOT Overview

several benefits. The first benefit is for application developers, who can re-use available transfer techniques instead of re-implementing them. The second benefit is for the inventors of innovative transfer techniques. Applications that use the transfer service can immediately begin using the new transfer techniques without modification. Innovative ideas do not need to be hacked into existing protocols using application-specific tricks. Because the transfer service sees the whole data object that the application wants to transfer, it can also apply techniques such as coding, multi-pass compression, and caching, that are beyond the reach of the underlying transport layers. The transfer service itself is not bound to using particular transports, or even established transports-it could just as well attempt to transfer the data using a different network connection or portable storage device.

Moving data transfer into a new service requires addressing three challenges. First, the service must provide a convenient and standard API for data transfer applications. Second, the architecture should allow easy development and deployment of new transfer mechanisms. Finally, the service must be able to support applications with diverse negotiation and naming conventions.

We present the design and implementation of a *Data-Oriented Transfer service*, or DOT, shown in Figure 1. The design of DOT centers around a clean interface to a modular, plugin based architecture to facilitate the adoption of new transfer mechanisms. DOT uses recent advances in content-based naming to name objects based upon their cryptographic hash, providing a uniform naming scheme across all applications.

This paper makes three contributions. First, we propose the idea of a data transfer service—a new way of structuring programs that do bulk data transfer, by separating their application-specific control logic from the generic function of data transfer. Second, we provide an effective design for such a service, its API and extension architecture, and its core transfer protocols. Finally, we evaluate our implementation of DOT with a number of micro- and macrobenchmarks, finding that it is easy to integrate with applications, and that by using DOT, applications can achieve significant bandwidth savings and easily take advantage of new network capabilities.

2 Transfer Service Scenarios

The advantage of a generic interface for data transfer is that it enables new transfer techniques across several applications. While we have implemented several transfer techniques within the DOT prototype, we believe its true power lies in the ability to accommodate a diverse set of scenarios beyond those in the initial prototype. This section examines several of these scenarios that we believe a transfer service enables, and it concludes with an examination of situations for which we believe the transfer service is inappropriate.

- A first benefit the transfer service could provide is cross-application caching. A DOT-based cache could benefit a user who receives the same file through an Instant Messaging application as well as via an email attachment. The benefits increase with multiuser sharing. An organization could maintain a single cache that handled all inbound data, regardless of which application or protocol requested it.
- Content delivery networks such as Akamai [1] could extend their reach beyond just the Web. A "data delivery network" could accelerate the delivery of Web, Email, NNTP, and any other data-intensive protocol, without customizing the service for each application. DOT could provide transparent access to Internet Backplane Protocol storage depots [4], to a storage infrastructure such as Open DHT [30], or to a collection of BitTorrent peers.
- The transfer service is not bound to a particular network, layer, or technology. It can use multi-path transfers to increase its performance. If future networks provided the capability to set up dedicated optically switched paths between hosts, the transfer service could use this facility to speed large transfers. The transfer need not even use the network: it could use portable storage to transfer data [14, 42].
- Finally, the benefits of the transfer service are not limited to simply exchanging bits. DOT creates the opportunity for making cross-application data processors that can interpose on all data transfers to and from a particular host. These proxies could provide services such as virus scanning or compression. Data processors combined with a delegation mechanism such as DOA [41] could also provide an architecturally clean way to perform many of the functions provided by today's network middleboxes.

The transfer service might be inappropriate for realtime communication such as telnet or teleconferencing. DOT's batch-based architecture would impose high latency upon such applications. Nor is the transfer service ideal for applications whose communication is primarily "control"

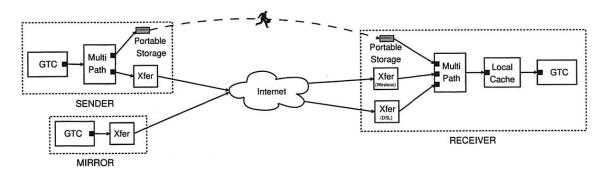


Figure 2: Example DOT Configuration. The GTC provides the transfer service functionality.

data, such as instant messaging with small messages. The overhead of the transfer service may wipe out any benefits it would provide. An important aspect of future work is to define this boundary more concretely—for instance, can the transfer service provide an interface to a multicast-based data transfer?

With these examples of the benefits that a transfer service could provide in mind, the next section examines a more concrete example of an example DOT configuration.

2.1 Example

Figure 2 shows an example DOT configuration that allows data transfers to proceed using multiple Internet connections and a portable storage device. This configuration provides three properties:

Surviving transient disconnection and mobility. Coping with disconnection requires that a transfer persist despite the loss of a transport-layer connection. In the example configuration, the multi-path plugin uses both the wireless and DSL links simultaneously to provide redundancy and load balancing. Mobility compounds the problem of disconnection because the IP addresses of the two endpoints can change. DOT's data-centric, content-based naming offers a solution to this problem because it is not tied to transport-layer identifiers such as IP addresses.

Transferring via portable storage. Portable storage offers a very high-latency, high-bandwidth transfer path [14, 42]. DOT's modular architecture provides an easy way to make use of such unconventional resources. A portable storage plugin might, for example, copy some or all of the object onto an attached disk. When the disk is plugged into the receiver's machine, the corresponding transfer plugin can pull any remaining data chunks from the disk. An advanced implementation might also make these chunks available to other machines in the network.

Using caching and multiple sources. The receiver can cache chunks from prior transfers, making them available to subsequent requests. The configuration above also shows how the transfer service can fetch data from multiple sources. Here, the multi-path plugin requests chunks in parallel from both the portable storage plugin and a set

of network transfer plugins. The transfer plugins pull data from two different network sources (the sender and a mirror site) over two network interfaces (wireless and DSL).

3 Related Work

There are considerable bodies of work that have explored better ways to accomplish data transfers and architectures that insert a protocol between the application and the transport layers. We believe that DOT differs from prior work in choosing an architectural split (running as a service and primarily supporting point-to-point object transfers) that is both powerful enough to support a diverse set of underlying mechanisms, and generic enough to apply to a wide variety of applications.

Our design for DOT borrows from content-addressable systems such as BitTorrent [8] and DHTs. Like DHTs, DOT uses content-based naming to provide an application-independent handle on data.

DOT also bears similarity to the Internet Backplane Protocol (IBP) [4], which aims to unify storage and transfer, particularly in Grid applications. Unlike IBP, DOT does not specify a particular underlying method for data transfer; rather, DOT separates transfer methods from applications, so that future protocols like IBP could be implemented and deployed more rapidly.

At the protocol level, BEEP, the Blocks Extensible Exchange Protocol [32], is close in spirit to DOT. BEEP aims to save application designers from re-inventing an application protocol each time they create a new application, by providing features such as subchannel multiplexing and capability negotiation on top of underlying transport layers. BEEP is a protocol framework, available as a library against which applications can link and then extend to suit their own needs. BEEP's scope covers the application's content negotiation and data transfer. In contrast, DOT is a *service* that is shared by all applications; thus, a single new DOT plug-in can provide new transfer mechanisms or interpose on data to all applications.

Protocols such as FTP [26], GridFTP [35], ISO FTAM (ISO 8571), and even HTTP [15] can be used by appli-

cations to access data objects, either by invoking a client for that protocol or by implementing it within the application protocol. Many of the transfer techniques that distinguish these protocols (e.g., GridFTP's use of parallel data streams or negotiation of transfer buffer sizes) could be implemented as a DOT transfer plugin. By doing so, an unmodified "DOT-based" FTP client would then be able to take advantage of the new functionality, reducing the effort required to adopt the protocol enhancements.

Proxies are commonly used to process legacy application traffic in new ways. While DOT aims to be more general than application-specific examples such as Web proxies, it bears resemblance to generic proxies such as the DNS-based OCALA [18] or the packet capture approaches used by RON [2] and the X-bone [38] to re-route traffic to an overlay. The drawback of these more generic approaches is that they lack knowledge of what the application is attempting to do (e.g., transfer a certain block of data) and so become limited in the tools they can apply. However, some of the advantages of DOT can be realized through the use of protocol-specific proxies. For example, our modified email server can be used as a mail relay/proxy when co-located with unmodified mail servers.

The initial DOT plugins borrow techniques from several research efforts. Rhea et al. designed a Web proxyto-proxy protocol that transfers content hashes to reduce bandwidth [29]. We show in Section 6.3 that DOT obtains similar benefits with email traffic. Spring and Wetherall use a similar hash-based approach to discover data duplication at the IP layer [34], and the rsync program uses a fingerprint-like approach to efficiently synchronize files across the network [39]. Mogul et al. showed similar benefits arising from using delta encoding in HTTP [22].

Finally, distributed object and file systems attempt to provide a common layer for implementing distributed systems. These systems range from AFS [17] and NFS [6] to research systems too numerous to cover in detail. Prominent among these are the storage systems that use content-addressable techniques for routing, abstracting identity, and saving bandwidth and storage [5, 9, 10, 12, 23, 27, 36]. Recent file systems have also incorporated portable storage for better performance [25, 37].

Like these distributed storage systems, DOT aims to mask the underlying mechanics of network data transfer from applications. Unlike these systems, DOT does not provide a single mechanism for performing the transfers. Its extensible architecture does not assume a "one size fits all" model for the ways applications retrieve their data. We do not wish to *force* the user of a DOT-based application to depend on an underlying distributed system if they only wish to perform point-to-point transfers. DOT complements many of these distributed systems by providing a single location where service developers can hook in, allowing applications to take advantage of their services.

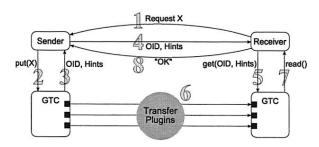


Figure 3: A data exchange using DOT

4 Design

This section first presents the basic system architecture, and then examines several details: (1) the manner in which applications use DOT; (2) the way the DOT transfer plugins and the default DOT transfer protocol operate; (3) the way that DOT accesses storage resources; and (4) the way that DOT plugins can be chained together to extend the system.

Applications interact with the transfer service as shown in Figure 1. Applications still manage their control channel, which handles content negotiation, but they offload bulk transfers to the transfer service. The transfer service delivers the data to the receiver using lower layer system and network resources, such as TCP or portable storage.

DOT is a receiver-pull, point-to-point service. We chose the receiver pull model to ensure that DOT only begins transferring data after both the sending and receiving applications are ready. We chose to focus on point-to-point service for two reasons: First, such applications represent *the* most important applications on the Internet (e.g., HTTP, email, etc.). Second, we believe that those point-to-multipoint applications that focus on bulk data transfer can be easily supported by DOT's plug-in model (e.g., transparent support for BitTorrent or CDNs).

Figure 3 shows the basic components involved in a DOTenabled data transfer: the sender, receiver, and the DOT transfer service. The core of the transfer service is provided by the DOT Generic Transfer Client (GTC). The GTC uses a set of *plugins* to access the network and local machine resources:

- Transfer plugins accomplish the data transfer between hosts. Transfer plugins include the default GTC-to-GTC transfer plugin and the portable storage transfer plugin.
- Storage plugins provide DOT with access to local data, divide data into chunks, and compute the content hash of the data. Storage plugins include the diskbacked memory cache plugin used in our implementation, or a plugin that accesses locally indexed data on the disk.

Sending data using DOT involves communication

among the sender, receiver, and the GTC. Figure 3 enumerates the steps in this communication:

- (1) The receiver initiates the data transfer, sending an application-level request to the sender for object X. Note that the sender could also have initiated the transfer.
- (2) The sender contacts its local GTC and gives the GTC object X using the put operation.
- (3) When the put is complete, the GTC returns a unique object identifier (OID) for X to the sender as well as a set of *hints*. These hints, described in Section 4.2.1, help the receiver know where it can obtain the data object.
- (4) The sender passes the OID and the hints to the receiver over the application control channel.
- (5) The receiver uses the get operation to instruct its GTC to fetch the object corresponding to the given OID.
- (6) The receiver's GTC fetches the object using its transfer plugins, described in Section 4.2, and then
 - (7) returns the requested object to the receiver.
- (8) After the transfer is complete, the receiver continues with its application protocol.

DOT names objects by OID; an OID contains a cryptographic hash of the object's data plus protocol information that identifies the version of DOT being used. The hash values in DOT include both the name of the hash algorithm and the hash value itself.

4.1 Transfer Service API

The application-to-GTC communication, shown in Table 1, is structured as an RPC-based interface that implements the **put** and **get** family of functions for senders and receivers respectively. A key design issue for this API is that it must support existing data-transfer applications while simultaneously enabling a new generation of applications designed from the ground-up to use a transfer service. Our experience implementing a DOT prototype has revealed several key design elements:

Minimal changes to control logic. Using the data transfer service should impose minimal changes to the control logic in existing applications. We created a stub library that provides a read/write socket-like interface, allowing legacy applications to read data from the GTC in the same way they previously read from their control socket. This interface requests that the GTC place data in-order before sending it to the application.

Permit optimization. The second challenge is to ensure that the API does not impose an impossible performance barrier. In particular, the API should not mandate extra data copies, and should allow an optimized GTC implementation to avoid unnecessarily hashing data to compute OIDs. Some applications also use special OS features (e.g., the zero-copy sendfile system call) to get high performance; the API should allow the use of such services or provide equivalent alternatives. To address such performance concerns, the GTC provides a file-descriptor

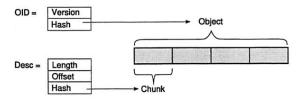


Figure 4: Relationship between DOT objects, chunks, OIDs and descriptors

passing API that sends data to the GTC via RPC.¹ DOT gives high-performance applications the choice to receive data out-of-order to reduce buffering and delays.

Data and application fate sharing. The third design issue is how long the sender's GTC must retain access to the data object or a copy of it. The GTC will retain the data provided by the application at *least* until either: (a) the application calls GTC_done(); or (b) the application disconnects its RPC connection from the GTC (e.g., the application has exited or crashed). There is no limit to how long the GTC may cache data provided by the application—our implementation retains this data in cache until it is evicted by newer data.

4.2 DOT Transfer Plugins

The GTC transfers data through one or more transfer plugins. These plugins have a simple, standard interface that can implement diverse new transfer techniques. Internally, the GTC splits objects into a series of *chunks*. Each chunk is named with a *descriptor*, which contains a hash of the chunk, its length, and its offset into the object. Figure 4 shows the relationship between objects, OIDs, chunks, and descriptors. The transfer plugin API has three functions:

```
get_descriptors(oid, hints[], cb)
get_chunks(descs[], hints[], cb)
cancel_chunks(descriptors[])
```

A transfer plugin must support the first two functions. The cb parameter is an asynchronous callback function to which the plugin should pass the returned descriptors or chunks. Some plugins may not be able to cancel an inprogress request once it has gone over the network, and so may discard cancel requests if necessary.

To receive data, the GTC calls into a single transfer plugin with a list of the required chunks. That plugin can transfer data itself or it can invoke other plugins. Complex policies and structures, such as parallel or multi-path transfers, can be achieved by a cascade of transfer plugins that build upon each other. For example, the current DOT prototype provides a multi-path plugin which, when

¹DOT does not yet cache object OIDs, but we note that systems such as EMC's Centera [13] already store the hash of files, so such an optimization is feasible.

Type	Command	Description
	PUT Com	mands
t_id	GTC_put_init()	Initiates an object "put". Returns a transaction identifier.
void	GTC_put_data(t_id, data)	Adds object data to the GTC
(OID, Hints)	GTC_put_commit(t_id)	Marks the end of object data transfer. Returns an opaque structure consisting of the OID and Hints
(OID, Hints)	GTC_put_fd(file descriptor)	Optimized put operation that uses an open file descriptor
void	GTC_done (OID)	Allows the GTC to release resources associated with OID.
	GET Com	mands
t_id	GTC_get_init(OID, mode, hints)	Initiates an object fetch. Returns a transaction identifier. Mode can be Sequential or Out-of-Order.
int	GTC_get(t_id, buf, &offset, &size)	Read data from the GTC. Returning zero indicates EOF.

Table 1: The Application-to-GTC API

instantiated, takes a list of other transfer plugins to which 4.3 Storage Plugins it delegates chunk requests.

Every GTC implementation must include a default GTC-GTC plugin that is available on all hosts. This plugin transfers data between two GTCs via a separate TCP connection, using an RPC-based protocol. The receiver's GTC requests a list of descriptors from the sender's GTC that correspond to the desired OID. Once it starts receiving descriptors, the receiver's GTC can begin sending pipelined requests to transfer the individual chunks.

4.2.1 Receiving data: Hints

As DOT is based on a receiver-pull model, hints are used to inform the receiver's GTC of possible data locations. They are generated by the sender's GTC, which the sender passes to the receiver over the application control channel. The receiver passes the hints to its GTC. Hints, associated with an OID, are only generic location specifiers and do not include any additional information on how the fetched data should be interpreted. Section 7 discusses how DOT could support mechanisms such as content encoding and encryption via additional per-chunk metadata.

A hint has three fields: method, priority, and weight. The method is a URI-like string that identifies a DOT plugin and then provides plugin-specific data. Some examples might be gtc://sender.example.com:9999/ or dht://OpenDHT/. As with DNS SRV records [16], the priority field specifies the order in which to try different sources; the weight specifies the probability with which a receiver chooses different sources with the same priority. By convention, the sender's GTC will include at least one hint-itself-because the sender's GTC is guaranteed to have the data.

The main purpose of the GTC is to transfer data, but the GTC must sometimes store data locally. The sender's GTC must hold onto data from the application until the receiver is able to retrieve it. The receiver's GTC, upon retrieving data might need to reassemble out-of-order chunks before handing the data to the receiver, or may wish to cache the data to speed subsequent transfers.

The GTC supports multiple storage plugins, to provide users and developers with flexible storage options. Examples of potential back-ends to the storage plugins include in-memory data structures, disk files, or an SOL database. The current DOT prototype contains a single storage plugin that uses in-memory hash tables backed by disk. The storage plugin is asynchronous, calling back to the GTC once it stores the data.

All DOT storage plugins provide a uniform interface to the GTC. To add data to the storage plugin, the GTC uses an API that mirrors the application PUT API in Table 1.

In addition to whole object insertion, the storage plugins export an API for single chunk storage. This API allows the GTC or other plugins to directly cache or buffer chunk data. The lookup functions are nearly identical to those supported by the transfer plugins, except that they do not take a hints parameter:

```
put_chunk(descriptor, data)
release_chunk(descriptor)
get_descriptors(oid, cb)
get_chunks(descriptors[], cb)
```

Configuring Plugins

DOT transfer plugins are configured as a data pipeline, passing get_descriptors and get_chunks requests on to subsequent transfer plugins. A simple DOT configuration consists of the GTC, a local storage plugin,

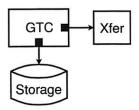


Figure 5: A simple DOT configuration

and a default GTC-GTC transfer plugin, shown in Figure 5. This configuration is instantiated as follows:

```
m = new gtcdMain();
sp = new storagePlugin(m);
xp = new xferPlugin(m);
m->set_xferPlugin(xp);
m->set_storagePlugin(sp);
```

Plugins that can push data, such as the portable storage plugin, need to be notified when new data becomes available. These plugins register with the GTC to receive notice when a new data object is inserted into the GTC for transfer, by calling register_push().

5 Implementation

DOT is implemented in C++ using the libasync [21] library. Libasync provides a convenient callback-based model for creating event-driven services. DOT makes extensive use of libasync's RPC library, libarpc, for handling communication both on the local machine and over the network. For instance, the default GTC-GTC protocol is implemented using RPC.

5.1 Multi-path Plugin

The DOT multi-path plugin acts as a load balancer between multiple transfer plugins, each of which is bound to a particular network interface on a multi-homed machine. The plugin is configured with a list of transfer plugins to which it should send requests. The sub-plugins can be configured in one of two ways. Some plugins can receive requests in small batches, e.g., network plugins that synchronously request data from a remote machine. Other plugins instead receive a request for all chunks at once, and will return them opportunistically. The latter mode is useful for plugins such as the portable storage plugin that opportunistically discover available chunks.

The multi-path plugin balances load among its subplugins in three ways. First, it parcels requests to subplugins so that each always has ten requests outstanding. Second, to deal with slow or failed links, it supports request borrowing where already-issued requests are shifted

from the sub-plugin with the longest queue to one with an empty queue. Third, it cancels chunk requests as they are satisfied by other sources.

5.2 Portable Storage Plugin

On the sender side, the portable storage plugin (PSP) registers to receive notifications about new OIDs generated by the GTC. When a new OID is discovered, the PSP copies the blocks onto the storage device. The implementation is naive, but effective: Each chunk is stored as a separate file named by its hash, and there is no index.

On the receiver, the PSP acts as a transfer plugin accessed by the multi-path plugin that receives a request for all descriptors. It polls the portable storage device every 5 seconds to determine if new data is available.² If the device has changed, the PSP scans the list of descriptors stored on the flash device and compares them to its list of requested descriptors, returning any that are available.

5.3 Chunking

The storage plugin also divides input data into chunks. To do so, it calls into a chunker library. The chunker is a C++ class that is instantiated for each data object sent to the storage plugin. It provides a single method, chunk_data, that receives a read-only pointer to additional data. It returns a vector of offsets within that data where the storage plugin should insert a chunk boundary. Our current implementation supports two chunkers, one that divides data into fixed-length segments, and one that uses Rabin fingerprinting [28, 20] to select data-dependent chunk boundaries. Rabin fingerprinting involves computing a function over a fixed-size sliding window of data. When the value of the function is zero, the algorithm signals a boundary. The result is that the boundaries are determined by the value of the data; they are usually the same despite small insertions or deletions of data.

5.4 Stub Library

Most applications are not written to send data using RPC calls to a local transfer service. To make it easy to port existing applications to use DOT, we created a socket-like stub library that preserves the control semantics of most applications. The library provides five functions:

```
dot_init_get_data(oid+hints)
dot_init_put_data()
dot_read_fn(fd, *buf, timeout)
dot_write_fn(fd, *buf, timeout)
dot_fds(*read, *write, *time)
```

get and put return a forged "file descriptor" that is used by the read and write functions. The read and

²Polling, while less efficient than receiving notification from the OS, was deemed more portable.

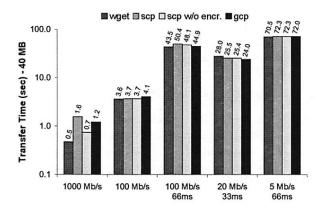


Figure 6: gcp vs. other standard file transfer tools

write functions provide a blocking read and write interface. The fds function allows select-based applications to query the library to determine which real file descriptors the application should wait for, and for how long. Section 6.3 shows how the stub library made it easy to integrate DOT with a production mail server package.

6 Evaluation

The primary goal of the DOT architecture is to facilitate innovation without impeding performance. This section first examines a number of microbenchmarks of basic DOT transfers to understand if there are bottlenecks in the current system, and to understand whether they are intrinsic or could be circumvented with further optimization. It then examines the performance of plugins that use portable storage and multiple network paths to examine the benefits that can arise from using a more flexible framework for data transfer. This section concludes by examining the integration of DOT into Postfix [40], a popular mail server.

6.1 Microbenchmarks

To demonstrate DOT's effectiveness in transferring bulk data, we wrote a simple file transfer application, gcp, that is similar to the secure copy (scp) program provided with the SSH suite. gcp, like scp, uses ssh to establish a remote connection and negotiate the control part of the transfer such as destination file path, file properties, etc. The bulk data transfer occurs via GTC-GTC communication.

We used this program to transfer files of sizes ranging from 4KB to 40MB under varying network conditions. In the interests of space, we present only the results from the 40MB file transfer as it highlights the potential overheads of DOT. These results, shown in Figure 6, compare the performance of gcp to wget, a popular utility for HTTP downloads, scp, and a modified version of scp that, like gcp, does not have the overhead of encryption for the bulk data transfer. All tools used the system's default values for

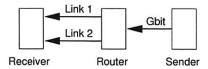


Figure 7: Topology for the multi-path evaluation. The capacities of links 1 and 2 are varied.

TCP's send and receive buffer size. gcp uses the fixed-size chunker for these experiments. All displayed results are the mean of 10 trials. All experiments were performed on a Gigabit Ethernet network at Emulab [43] with a dummynet [31] middlebox controlling network bandwidth and latency. The machines were Emulab's "pc3000" nodes with 3GHz 64-bit Intel® Xeon® processors and 2GB of DRAM.

For WAN conditions, gcp exhibits very little or no overhead when compared to the other file transfer tools and its performance is equivalent to scp both with and without encryption. On the local Gigabit network, gcp begins to show overhead. In this case, wget is the fastest. Unlike both scp and gcp, the Apache HTTP server is always running and wget does not pay the overhead of spawning a remote process to accept the data transfer. gcp is only slightly slower than scp. This overhead arises primarily because the GTC on the sender side must hash the data twice: once to compute the OID for the entire object and once for the descriptors that describe parts of the object. The hashing has two effects. First, the computation of the whole data hash must occur before any data can be sent to the receiver, stalling the network temporarily. Second, the hashes are relatively expensive to compute.

This overhead can be reduced, and the network stall eliminated, by caching OIDs (as noted earlier, some systems already provide this capability). While the computational overhead of computing the chunk hashes remains, it can be overlapped with communication. The computational overhead could also be reduced by generating a cheaper version of the OID, perhaps by hashing the descriptors for the object. However, the latter approach sacrifices the uniqueness of the OID and removes its usefulness as an end-to-end data validity check; we believe that retaining the whole-data OID semantics is worthwhile.

6.2 Plugin Effectiveness

This section examines the performance of the two transfer plugins we created for DOT, the multi-path plugin and the portable storage plugin.

6.2.1 Multi-Path Plugin

The multi-path plugin, described in Section 5.1, load balances between multiple GTC-GTC transfer plugins. We evaluate its performance using the same Emulab nodes as

Link 1	Link 2	single	multipath	savings
100/0	100/0	3.59	1.90	47.08%
100/0	10/0	3.59	3.54	1.39%
	100/33	21.46	11.15	48.04%
100/33	10/33	21.46	13.58	36.72%
	1/33	21.46	20.44	4.75%
	100/66	43.33	23.20	46.46%
100/66	10/66	43.33	22.97	46.99%
	1/66	43.33	38.25	11.72%
	10/66	48.39	23.42	51.60%
10/66	1/66	48.39	39.20	18.99%
	0.1/66	48.39	44.14	8.78%
1/66	0.1/66	367.39	313.42	14.69%

Table 2: Multi-Path evaluation results. Links indicate bandwidth in Mbit/s and latency in milliseconds. The single column shows the time for a gcp transfer using only the fastest link of the pair.

above. The receiver is configured with two network interfaces of lower capacity, and the sender with one Gigabit link, as shown in Figure 7.

Like the microbenchmarks, we examined the performance of the multi-path plugin in transferring 40MB, 4MB, and 400KB files via gcp. For brevity, we present only the 40MB results, but note that the performance on 400KB files was somewhat lower because they were not sufficiently large to allow TCP to consume the available capacity. All transfers were conducted using FreeBSD 5.4 with the TCP socket buffers increased to 128k and the initial slow-start flight size set to 4 packets.

Table 2 presents several combinations of link bandwidths and latencies, showing that the multi-path plugin can substantially decrease transfer time. For example, when load balancing between two directly connected 100 Mbit/s Ethernet links, the multi-path plugin reduced the transfer time from 3.59 seconds on a single link to 1.90 seconds. The best possible time to transfer a 40MB file over 100 Mbit/s Ethernet is 3.36 seconds, so this represents a substantial improvement over what a single link could accomplish. We note that the multi-path plugin was created and deployed with no modifications to gcp or the higher layer DOT functions.

In high bandwidth×delay networks, such as the 100 Mbit/s link with 66ms latency (representing a high-speed cross-country link), TCP does not saturate the link with a relatively small 40MB transfer. In this case, the benefits provided by the multi-path plugin are similar to those achieved by multiple stream support in GridFTP and other programs. Hence, using the multi-path plugin to bond a 100 Mbit/s and 10 Mbit/s link produces greater improvements than one might otherwise expect.

Finally, in cases with saturated links with high asymmetry between the link capacities (the second to last line in the table, a 10 Mbit/s link combined with a 100 Kbit/s link),

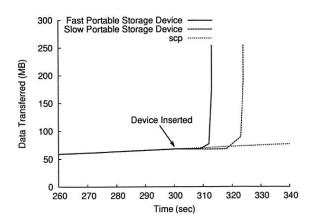


Figure 8: Portable Storage performance. The receiving machine completed the transfer from the USB flash device once it was inserted after 300 seconds. Without the flash device, the scp transfer took 1126 seconds to complete.

the multi-path plugin provides some benefits by adding a second TCP stream, reducing the transfer time by roughly 9%. To understand the impact of request borrowing, we disabled it and re-ran the same experiment. Without request borrowing, the multi-path plugin slowed the transfer down by almost a factor of three, requiring 128 seconds to complete a transfer that the single link gcp completed in 48 seconds. Without request borrowing, the queue for the fast link empties, and the transfer is blocked at the speed of the slowest link until its outstanding requests complete.

6.2.2 Portable Storage

We evaluate DOT's use of portable storage using a work-load drawn from *Internet Suspend/Resume (ISR)* [33]. ISR is a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed storage system.

A key ISR challenge is in dealing with the large VM state, typically many tens of GB. When a user suspends a ISR machine, the size of *new* data generated during the session is in the order of hundreds of MBs. This includes both the changes made to the virtual disk as well as the serialized copy of the virtual memory image. Given the prevalence of asymmetric links in residential areas, the time taken to transfer this data to the server that holds VM state can be substantial. However, if a user is willing to carry a portable storage device, part of the VM state can be copied to the device at suspend time.

To evaluate the benefit of DOT in such scenarios, we simulate a fast residential cable Internet link with a maximum download and upload speed of 4 Mbit/s and 2 Mbit/s respectively. An analysis of data collected from the ISR

test deployment on Carnegie Mellon's campus [24] revealed that the average size of the data transferred to the server after compression is 255 MB. To model this scenario, we use DOT to transfer a single 255 MB file representing the combined checkin state. A USB key chain is inserted into the target machine³ approximately five minutes after the transfer is initiated. The test uses two USB devices, a "fast" device (approx. 20MB/sec. read times) and a "slow" device (approx. 8MB/sec).

The speed of the transfer and total time for competition is presented in Figure 8. Immediately after the portable storage is inserted, the system experiences a small reduction in throughput as it scans the USB device for data.⁴ As it starts reading the data from the device, the transfer rate increases substantially. Once the data has been read and cached, the transfer completes almost instantly.

6.3 Case Study: Postfix

To evaluate the ease with which DOT integrates with existing, real networked systems, we modified the Postfix mail program to use DOT, when possible, to send email messages between servers. Postfix is a popular, full-featured mail server that is in wide use on the Internet, and represented a realistic integration challenge for DOT.

We chose to examine the benefits of running DOT on a mail server for a number of reasons. First, mail is a border-line case for a mechanism designed to facilitate large data transfers. Unlike peer-to-peer or FTP downloads, most mail messages are small. Mail provides an extremely practical scenario: the servers are complex, the protocol has been around for years and was not designed with DOT in mind, and any benefits DOT provides would be beneficial to a wide array of users.

Postfix is a modular email server, with mail sending and receiving decomposed into a number of separate programs with relatively narrow functions. Outbound mail transmission is handled by the *smtp* client program, and mail receiving is handled by the *smtpd* server program. Postfix uses a "process per connection" architecture in which the receiving demon forks a new process to handle each inbound email message. These processes use a series of blocking calls, with timeouts, to perform network operations. Error handling is controlled via a setjmp/longjmp exception mechanism.

DOT was integrated into this architecture as an SMTP protocol extension. All DOT-enabled SMTP servers, in addition to the other supported features, reply to the EHLO greeting from the client with a "X-DOT-DATA" response. Any SMTP client compliant with the RFC [19] can safely ignore unrecognized SMTP options beginning with "X".

This allows non-DOT enable clients to use the standard method of data transfer and allows the server to be backward compatible.

On the presentation of X-DOT-DATA by the server, any DOT-enabled client can use the X-DOT-DATA command as a replacement for the "DATA" command. Clients, instead of sending the data directly to the server, only send the OID and hints to the server as the body of the X-DOT-DATA command. Upon receipt of these, the server opens a connection to its local GTC and requests the specified data object. The server sends a positive completion reply only after successfully fetching the object. In the event of a GTC error, the server assumes that the error is temporary and a *transient* negative completion reply is sent to the client. This will make sure that the client either retries the DOT transfer or, after a certain number of retries, falls back to normal DATA transmission.

6.3.1 Mail Server Trace Analysis

The first results in this section are analytical results from a mail trace taken on a medium-volume research group mail server. This analysis serves two purposes. First, we examine the benefits of DOT's content-hash-based chunked encoding and caching. Chunked encoding is well known to benefit bulk peer-to-peer downloads [8] and Web transfers [29]; we wished to demonstrate that these benefits extend to an even wider range of applications. The second purpose is to generate a trace of email messages with which to evaluate our modified Postfix server.

Each message in the trace records an actual SMTP conversation with the mail server, recorded by a Sendmail mail filter. The traces are anonymized, reporting the keyed hash (HMAC) of the sender, receiver, headers, and message body. The anonymization also chunks the message body using a static sized chunk and Rabin fingerprints and records the hashes of each chunk, corresponding to the ways in which DOT could chunk the message for transmission. The subsequent analysis examines how many of these chunks DOT would transmit, but does *not* include the overhead of DOT's protocol messages. These overheads are approximately 100 bytes per 20KB of data, or 0.5%, and are much smaller than the bandwidth savings or the variance between days.

The email workload was an academic research group, with no notable email features (such as large mailing lists). We hypothesize that the messages in this workload are somewhat smaller than they would be in a corporate environment with larger email attachments, but we lack sufficient data about these other environments. The mail server handles approximately 2,500 messages per day. The traces cover 458,861 messages over 159 days. The distribution of email sizes, shown in Figure 9, appears heavy-tailed and consistent with other findings about email distribution. The sharp drop at 10-20MB represents common cut-off values

³Both machines have 3.2GHz Intel® Pentium® 4 CPUs with 2GB of SDRAM and run the 2.6.10-1.770-SMP Linux kernel.

⁴This slowdown could be avoided by spawning a helper process to perform the disk I/O.

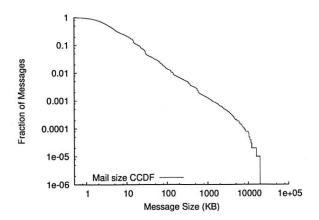


Figure 9: Mail message size distribution follows a heavytailed distribution until message size limits are reached.

for allowed message sizes. We eliminated one message that was 239MB, which we believe was a test message sent by the server administrator.

This section examines four different scenarios. SMTP default examines the number of bytes sent when sending entire, unmodified messages. With DOT body, the mail server sends the headers of the message directly, and uses DOT to transfer the message body. Only whole-file caching is performed: either the OID has been received, or not. With Rabin body, the mail server still sends the headers separately, but uses Rabin fingerprinting to chunk the DOT transfer. Finally, Rabin whole sends the entire message, headers included, using DOT. Because the headers change for each message, sending a statically-chunked OID for the entire message is unlikely to result in significant savings. The Rabin whole method avoids this problem, at the cost of some redundancy in the first content block. Allowing the GTC to chunk the entire message also allows the simplest integration with Postfix by avoiding the need for the mail program to parse the message content when sending. Our analysis assumes that DOT is enabled on both SMTP clients and servers. Table 3 shows the number of bytes sent by DOT in these scenarios.

In all, DOT saves approximately 20% of the total message bytes transferred by the mail server. These benefits arise in a few ways. As can be seen in Figure 10, a moderate number of messages are duplicated exactly once, and a small number of messages are duplicated many times—nearly 100 copies of one message arrived at the mail server. Second, as Table 3 showed, there is considerable partial redundancy between email messages that can be exploited by the Rabin chunking.

While our study did not include the number of large email attachments that we believe are more common in corporate environments, we did observe a few such examples. 1.5% of the bytes in our trace came from a 10 MByte email that was delivered eleven times to local users. The administrator of the machine revealed that a non-local

Method	Total Bytes	Percent Bytes
SMTP default	6800 MB	-
DOT body	5876 MB	86.41 %
Rabin body	5056 MB	74.35 %
Rabin whole	5496 MB	80.81 %

Table 3: Savings using different DOT integration methods. This table does not include DOT overhead bytes.

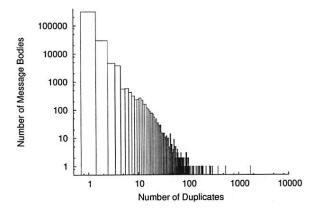


Figure 10: Histogram of the number of repetitions of particular message bodies. There were 307,969 messages with unique bodies, 29,746 with a body duplicated once, and so on. One message was duplicated 1,734 times.

mailing list to which several users were subscribed had received a message with a maximum-sized email attachment. Such incidents occur with moderate frequency, and are a common bane for email administrators. By alleviating the bandwidth, processing, and storage pain from such incidents, we hope that DOT can help allow users to communicate in the way *most convenient to them*, instead of following arbitrary decrees about application suitability.

6.3.2 System throughput

To evaluate DOT's overhead, we generated 10,000 email messages from beginning of the mail trace. Each message is the same size as a message from the trace, and the message's content is generated from Rabin-fingerprint generated hash blocks in the trace message. Each unique hash value is deterministically assigned a unique content chunk by seeding a pseudo-random number generator with the hash value. The generated emails preserve some of the similarity between email messages, but because we cannot regenerate the original content (or other content that contains the same Rabin chunk boundaries), the generated emails are somewhat less redundant than the originals.

We replayed the 10,000 generated messages through Postfix running on the same machines used for the portable storage evaluation, connected with 100 Mbit/s Ethernet. Table 4 shows that the DOT-enabled Postfix required only

Program	Seconds	Bytes sent
Postfix	468	172 MB
Postfix-DOT	468	117 MB

Table 4: Postfix throughput for 10,000 email messages

Program	Original LoC	New LoC	%
GTC Library	-	421	-
Postfix	70,824	184	0.3%
smtpd	6,413	107	1.7%
smtp	3,378	71	2.1%

Table 5: Lines of Code Added or Modified in Postfix

68% of the total bandwidth, including all protocol overhead, but both systems had identical throughput.

The Postfix workload is extremely disk-seek bound. Neither the considerable bandwidth savings or the slight CPU overhead provided by DOT was sufficient to change this. We therefore believe that the widespread adoption of DOT in mail servers would therefore have a positive effect on bandwidth use without imposing noticeable overhead.

6.3.3 Integration Effort

Integrating DOT with Postfix took less than a week for a graduate student with no knowledge of Postfix. This time includes the time required to create the adapter library for C applications that do not use libasync, discussed in Section 5.4. Table 5 presents the number of lines of code (LoC) needed to enable DOT within Postfix. The modifications touched two Postfix subsystems, the smtpd mail server program, and the smtp mail client program. While we have only used two applications thus far with DOT (Postfix and gcp), we are encouraged by the ease with which DOT integrated with each.

7 Discussion

In previous sections, we presented the benefits we have observed using our initial DOT implementation, and discussed some of the benefits we believe DOT can realize from a transfer service in other scenarios. Our experience with DOT thus far has revealed several lessons and remaining challenges.

Efficiency. Our design of the DOT default transfer protocol assumes that its extra round trips can be amortized across file transfers of sufficient length. While this design is effective for DOT's intended target of large transfers, we would like to make the transfer service efficient for as many applications as possible, even if their transfers are small. For example, highly interactive Web services such as Google go to considerable effort to reduce the number of round-trips experienced by clients, and providers such as Akamai tout such reduction as a major benefit of their

service. As the email study in Section 6.3 noted, email and many other protocols have a heavy tailed distribution with numerous small files being transferred.

There are two likely approaches that can reduce DOT's overhead for small transfers: either allow the application to send small objects directly, or implement a transfer plugin that passes small objects in-line as a hint over its control channel to bypass the need for additional communication. For reasons discussed below, we believe the former may be a better approach.

Application resilience. The DOT-enabled Postfix falls back to normal SMTP communication if either the remote host does not support DOT or if it cannot contact the GTC running on the local machine. This fallback to direct transfer makes the resulting Postfix robust enough to run as the primary mail program for one of our personal machines. As Cappos and Hartman noted on Planetlab, applications that depend on cutting-edge services are wise to fall back to simpler and more reliable mechanisms [7]. We believe this is sound advice for any application making use of DOT.

Security. In a system such as DOT, applications have several choices that trade privacy for efficiency. The simplest way for an application to ensure privacy is to encrypt its data before sending it to the transfer service. Unfortunately, this places restrictions on the transfer service's ability to cache data and obtain data from independent sources. Another simple (and common) choice is to not encrypt the data at all, and trust in whatever host or network security mechanisms are available. This approach allows the transfer service the most opportunities to exploit features of the data to improve transfers, but offers little security to the user. A promising middle ground is to support convergent encryption [11], in which data blocks are encrypted using their own hash values as keys. Only recipients that know the hashes of the desired data can decrypt the data blocks.

DOT must interact with many different applications, each of which may have its own mechanisms and policies for providing security and privacy. We feel strongly that a transfer service must support a wide variety of application requirements, and should not impose a particular security model or flavor of cryptography or key management. However, an efficient implementation of convergent encryption requires some support from the transfer service: The algorithm that splits data into chunks must be consistent across implementations, or the resulting encrypted blocks cannot be effectively cached. We intend to support convergent encryption in the near future.

Application preferences and negotiation. The current implementation of DOT performs data transfers completely independently of the application. While this suffices for a large and important class of applications, we believe that DOT can also benefit applications that desire more control over how their transfers are effected. Examples of such applications include those that have specific encryption requirements or that wish to take advantage of

different sending rates or quality of service categories.

Providing convergent encryption illustrates some of the problems that we must address. Applications that require encryption must be able to inform the transfer layer of their needs, and confirm that the data *will* be encrypted properly. Furthermore, the sender and receiver GTCs must also negotiate the availability of specific capabilities in case they are running different versions of DOT or have different plugins. Finally, the convergent encryption plugins must communicate the encryption keys.

The design of capability discovery and negotiation is an important part of our future work. As a first cut at supporting plugins that must negotiate specific capabilities, we are adding two types of metadata to DOT: per-object metadata that allows the receiving side to construct the plugin-chain required to process the data; and per-chunk metadata that allows the plugins to send information (such as encryption keys) needed to process the chunks when they are received.

Selecting Plugins and Data Sources. A DOT receiver may be able to obtain its desired data from multiple sources using several different plugins. The DOT architecture uses a hierarchy of plugins, some of which implement selection policies, to arbitrate between sources. If a variety of plugins become available, DOT's plugin configuration interface may need to evolve to support more sophisticated configurations that allow plugins to determine the capabilities of their upstream and downstream plugins. For example, it may be advantageous for a multi-path plugin to know more about the capacities of attached links or storage devices. While we believe such an interface could be useful, its design would be purely speculative in the absence of a wide variety of plugins from which to choose.

Several of our plugins represent promising starts more than finished products. We plan to enhance the multi-path plugin to support fetching data from mirror sites and multi-homed servers as well as multi-homed receivers. We are beginning the initial design for a "rendezvous" service that allows portable storage to be plugged in to a third party machine, instead of directly to the receiver.

Supporting dynamically generated objects. Our design calls for DOT to handle dynamically generated objects that cannot be fully hashed before transmission by assigning them a random OID. This change requires a small modification to the API to return the OID before the put call has completed, and requires that the remote get_descriptors call be able to return an indicator that the receiver should continue checking for more descriptors. A drawback is that random OIDs sever the tie between the OID and the data contents, which prevents per-object (but not per-chunk) caching.

Exploiting structure in data. A final issue in the design of DOT is the division of labor between the application and the transfer service in dividing data into chunks. In our design, the transfer service is solely responsible for chunking. As our evaluation of email traces showed, the use of

Rabin fingerprinting can remove the need for applicationspecific chunking decisions in some cases, but there may remain other cases in which application knowledge is helpful. For instance, many applications transfer structured data. Databases, for example, may return data that has repetitions at the row level, which may be much smaller than DOT's default chunk size. While our current techniques work well for email and Web objects, we believe this issue merits further exploration.

8 Conclusion

This paper presented the design and implementation of an extensible data-oriented transfer service, DOT. DOT decouples application-specific content negotiation from the more general process of transferring data across the network. Using such a transfer service reduces reimplementation at the application layer and facilitates the adoption of new technologies to improve data transfer. Through a set of microbenchmarks and an examination of a production mail server modified to use DOT, we have shown that the DOT architecture imposes little overhead. DOT provides significant benefits, reducing bandwidth use and making new functionality such as multi-path or portable storage-based transfers readily available.

While our design still faces several challenges, we believe that introducing data transfer as a system *service* is a worthwhile goal. A widely deployed transfer service helps in the evolution of new services: researchers could easily try out new protocols using real, unmodified applications; and a significant fraction of Internet traffic could make use of new network-layer functions by simply adding a new transfer plugin. We believe that DOT could provide significant benefits to applications, networks, and ultimately, to users.

Acknowledgments

We are grateful to Hari Balakrishnan, Nick Feamster, Michael Freedman, our shepherd John Hartman, Larry Huston, Brad Karp, Dina Katabi, Mahim Mishra, M. Satyanarayanan, Alex Snoeren, Eno Thereska, and our anonymous reviewers for their valuable feedback. We thank Emulab for providing resources for evaluating DOT. This research was supported by NSF CAREER award CNS-0546551, by NSF grant CNS-0509004, by Intel Research, and by a grant from the Carnegie Mellon CyLab.

References

- [1] Akamai. http://www.akamai.com, 1999.
- [2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct. 2001.

- [3] N. B. Azzouna and F. Guillemin. Analysis of ADSL traffic on an ip backbone link. In *Proc. IEEE Conference on Global Communica*tions (GlobeCom), San Francisco, CA, Dec. 2003.
- [4] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.
- [5] T. C. Bressoud, M. Kozuch, C. Helfrich, and M. Satyanarayanan. OpenCAS: A flexible architecture for content addressable storage. In 2004 International Workshop on Scalable File Systems and Storage Technologies, San Francisco, CA, Sept. 2004.
- [6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification, June 1995. RFC 1813.
- [7] J. Cappos and J. Hartman. Why it is hard to build a long-running service on PlanetLab. In *Proc. Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2005.
- [8] B. Cohen. Incentives build robustness in BitTorrent. In Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 2003.
- [9] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.
- [10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Canada, Oct. 2001.
- [11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. 22nd Intl. Conf on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [12] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, pages 75–80, Schloss-Elmau, Germany, May 2001.
- [13] EMC Centera Content Addressed Storage System. EMC Corporation, 2003. http://www.emc.com/.
- [14] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. ACM SIGCOMM*, pages 27–34, Karlsruhe, Germany, Aug. 2003.
- [15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Internet Engineering Task Force, Jan. 1997. RFC 2068.
- [16] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). Internet Engineering Task Force, Feb. 2000. RFC 2782.
- [17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems, 6 (1), February 1988.
- [18] J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. Supporting legacy applications over i3. Technical Report UCB/CSD-04-1342, University of California, Berkeley, May 2004.
- [19] J. Klensin. Simple Mail Transfer Protocol. Internet Engineering Task Force, Apr. 2001. RFC 2821.
- [20] U. Manber. Finding similar files in a large file system. In Proc. Winter USENIX Conference, pages 1–10, San Francisco, CA, Jan. 1994.
- [21] D. Mazières, F. Dabek, E. Peterson, and T. M. Gil. Using libasync. http://pdos.csail.mit.edu/6.824-2004/doc/libasync.ps.
- [22] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In Proc. First Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, Mar. 2004.
- [23] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

- [24] P. Nath, M. Kozuch, D. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proc. USENIX Annual Techincal Conference*, Boston, MA, June 2006. To appear.
- [25] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. 6th USENIX OSDI*, pages 363–378, San Francisco, CA, Dec. 2004.
- [26] J. B. Postel and J. Reynolds. File Transfer Protocol (FTP). Internet Engineering Task Force, Oct. 1985. RFC 959.
- [27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies* (FAST), pages 89–101, Monterey, CA, Jan. 2002.
- [28] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [29] S. C. Rhea, K. Liang, and E. Brewer. Value-based web caching. In Proc. Twelfth International World Wide Web Conference, Budapest, Hungary, May 2003.
- [30] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.
- [31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. ACM Computer Communications Review, 27(1): 31–41, 1997.
- [32] M. Rose. On the Design of Application Protocols. Internet Engineering Task Force, Nov. 2001. RFC 3117.
- [33] M. Satyanaranyanan, M. A. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. Pervasive and Mobile Computing, 1(2):157–189, 2005.
- [34] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIG-COMM*, Stockholm, Sweden, Sept. 2000.
- [35] The Globus Project. GridFTP: Universal data transfer for the Grid. http://www-fp.globus.org/datagrid/deliverables/C2WPdraft3.pdf, Sept. 2000.
- [36] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proc. USENIX Annual Technical Con*ference, pages 127–140, San Antonio, TX, June 2003.
- [37] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proc. 3rd USENIX Conference* on File and Storage Technologies, San Francisco, CA, Mar. 2004.
- [38] J. Touch and S. Hotz. The X-Bone. In Proc. 3rd Global Internet Mini-Conference in conjunction with IEEE Globecom, pages 75– 83, Sydney, Australia, Nov. 1998.
- [39] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.
- [40] W. Venema. The Postfix home page. http://www.postfix.org/.
- [41] M. Walfish, J. Stribling, and M. Krohn. Middleboxes no longer considered harmful. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.
- [42] R. Y. Wang, S. Sobti, N. Garg, E. Ziskind, J. Lai, and A. Krishnamurthy. Turning the postal system into a generic digital communication mechanism. In *Proc. ACM SIGCOMM*, pages 159–166, Portland, OR, Aug. 2004.
- [43] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc.* 5th USENIX OSDI, pages 255–270, Boston, MA, Dec. 2002.

OCALA: An Architecture for Supporting Legacy Applications over Overlays

Dilip Joseph¹ Karthik Lakshminarayanan¹ Jayanth Kannan¹ Ion Stoica¹ Ayumu Kubota² Klaus Wehrle³

¹University of California at Berkeley ²KDDI Labs ³RWTH Aachen University

Abstract

In order for overlays and new network architectures to gain real user acceptance, users should be able to leverage overlay functionality without any modifications to their applications and operating systems. We present our design, implementation, and experience with OCALA, an overlay convergence architecture that achieves this goal. OCALA interposes an overlay convergence layer below the transport layer. This layer is composed of an overlay independent sub-layer that interfaces with legacy applications, and an overlay dependent sub-layer that delivers packets to the overlay. Unlike previous efforts, OCALA enables: (a) simultaneous access to multiple overlays (b) communication between hosts in different overlays (c) communication between overlay hosts and legacy hosts (d) extensibility, allowing researchers to incorporate their overlays into OCALA. We currently support five overlays, i3 [32], RON [1], HIP [19], DOA [39] and OverDoSe [31] on Linux, Windows XP/2000 and Mac OS X. We (and a few other research groups and end-users) have used OCALA for over a year with many legacy applications ranging from web browsers to remote desktop applications.

1 Introduction

Over the past two decades, researchers have proposed a plethora of solutions to extend the Internet's functionality, and to improve its resilience and security. After sustained efforts to add new functions such as mobility [25] and multicast [5] to IP, researchers have recently turned their attention to developing new network architectures (e.g., [1,3,4,19,29,32,35]) and using overlays to address the Internet's limitations. This trend has been fueled by the difficulty of changing IP, on one hand, and by the advent of the PlanetLab [26] testbed and the recent NSF GENI [23] initiative—which promises to create a worldwide testbed for evaluating new network architectures—on the other hand.

In order to evaluate the feasibility of these proposals and to ultimately bring them closer to reality, it is important to experiment with real users running real applications. Ideally, users should be able to opt into new experimental architectures without any changes to their legacy applications. (We use the term legacy applications to refer to existing applications like web browsers that assume IP semantics.) Supporting legacy applications on new network architectures is inherently a difficult task: legacy applications assume traditional semantics of IP addresses and DNS names, while a new network architecture may offer a substantially different interface. Existing solutions are in general tailored to a particular network architecture [1,19,34,43], leading to duplication of effort across different implementations.

In this paper, we describe the design and implementation of our solution, OCALA (Overlay Convergence Architecture for Legacy Applications), that enables legacy applications to take advantage of the functionality provided by new network architectures. OCALA differs from existing solutions in that it enables (1) applications running on the same machine to access different overlays simultaneously, (2) stitching of multiple overlays so that users residing in different overlays can communicate with each other, (3) hosts to communicate through an overlay even if the other end-point understands only IP, and (4) extensibility so that a new overlay can be incorporated into OCALA with minimal effort.

In a nutshell, OCALA re-factors the protocol stack by imposing an *Overlay Convergence* (OC) layer. The OC layer is positioned below the transport layer in the IP stack. It is decomposed into the overlay-independent (OC-I) sub-layer, which interacts with the legacy applications by presenting an IP-like interface, and the overlay-dependent (OC-D) sub-layer, which tunnels the traffic of applications over overlays.

The main contributions of this paper are an overlay agnostic architecture for supporting legacy applications and an extensible implementation of this architecture as a proxy. Our implementation of OCALA as a proxy requires no changes to applications or operating systems.

In realizing our design, we borrow many techniques and protocols from the literature, such as address virtu-

¹We focus on the interface provided by the network substrate, and not on how this substrate is implemented. Thus, we do not distinguish between the implementation of a network architecture and an overlay; an overlay is one way to implement a new network architecture on top of IP.

alization [12, 19, 34, 36, 42, 43], DNS capture and rewriting [9, 22, 28, 42], and SSL [10]. We have implemented the OC-D sub-layer for *i*3 and RON. In addition, OC-D modules for HIP [14], DOA [39] and OverDoSe [31] have been implemented by other research groups. To illustrate the utility of OCALA, we have provided services such as intrusion-detection, secure wireless access, secure Intranet access, and Network Address Translation (NAT) box traversal, to legacy applications.

OCALA does not come without limitations. The fact that OCALA is positioned below the transport layer makes it hard, if not impossible, for legacy applications to take advantage of network architectures that provide transport or application layer functionalities, such as multi-path congestion control and data storage [17].

2 Related Work

Supporting legacy applications over non-IP or IP-modified communication infrastructures has been addressed in many contexts. Examples include overlay networks and new network architectures (e.g., RON [1], i3 [32], HIP [19], DOA [39], WRAP [2], end-host support for mobility [34, 36, 42]), and mechanisms that enable end-hosts to use overlays without participating in them [21]. In contrast to these overlay-specific efforts, OCALA enables a user to simultaneously access different overlays and to communicate with hosts residing in overlays the user is not directly connected to.

A recent system, Oasis [18], enables legacy applications to route traffic through different overlays, and has a design similar to that of OCALA. However, Oasis's current implementation supports only overlays that are IP-addressable, and does not allow stitching together multiple overlays. In contrast, OCALA supports overlays that address hosts using a variety of identifiers and naming schemes (e.g., i3, DOA), and allows hosts on different overlays to communicate with each other. Oasis optimizes application performance by automatically selecting the "best" overlay. Furthermore, Oasis supports sand-boxed code execution, a direct result of its Java-based implementation. The current implementation of OCALA does not support any of these features.

Our goal of stitching together multiple network architectures resembles the goal of AVES [22], TRIAD [3], UIP [7], IPNL [8], Plutarch [4], and IPv4/IPv6 transition schemes like [11]. In contrast to these proposals which provide universal connectivity, OCALA's focus is on exposing to users, functions that new architectures provide, both in isolation and when stitched together.

Layering is a widely-used principle in networking. Many architectures (e.g., HIP [19], WRAP [2]) hide the details of underlying layers by interposing a shim layer between the transport and network layers. More recently, Henderson generalized the HIP setup protocol to support

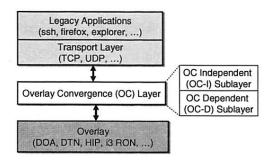


Figure 1: The overlay convergence (OC) layer.

other architectures that decouple location and identity of hosts [13]. OCALA's OC layer is similar to such a shim layer. OCALA is different from other architectures in that it explicitly splits the OC layer into an overlay independent sub-layer and an overlay dependent sub-layer, which respectively act as traditional network and link layers. This division enables OCALA to provide simultaneous access to and inter-operability across different network architectures.

For implementing OCALA, we rely on techniques and protocols previously proposed in different contexts. Intercepting DNS requests for interposing proxies on the data path has been used in AVES [22], Coral [9], and for improving web browsing performance over wireless networks [28]. Local-scope addresses have been utilized in the context of supporting mobility [19, 34, 36, 42], redirection [12], process migration [33, 34] and server availability [33]. Our address-negotiation protocol is similar to that in Yalagandula *et. al.* [42], while the OC-I sublayer's security protocol is a generalization of SSL [10].

3 Design Overview

We focus on network architectures and overlays that offer a service model of end-to-end packet delivery similar to IP's, as opposed to those that provide transportor application-layer functions, such as data storage (e.g., Oceanstore [17]). Some examples are overlays that improve Internet's resilience and performance (e.g., RON [1], Detour [29], OverQoS [35]), overlays that provide new functions (e.g., mobility [42,43]), overlays that bridge multiple address spaces [2,22], as well as recent architectures such as i3 [32], HIP [19], and DOA [39]. Although not all architectures are realized as overlays, for convenience, in the remainder of this paper, we will use the term overlay to also refer to network architectures that are implemented as overlays.

Each end-host E in an overlay has an overlay-specific identifier (ID), which is used by other end-hosts to contact E through the overlay. While in the simplest case an overlay ID can be the host's IP address (e.g., RON), many overlays use other forms of identifiers (e.g., i3 and DOA use flat IDs, HIP uses hashes of public keys). Since

overlay IDs may not be human-readable, end-hosts may also be assigned human-readable names for convenience.

3.1 Goals

Our design is centered around four main goals:

- Transparency: Legacy applications should not break despite the fact that their traffic is relayed over an overlay instead of over IP.
- Inter-operability: Hosts in different overlays should be able to communicate with one another, and further, hosts that do not participate in any overlay should also be accessible through overlays.
- Expose overlay functionality: Users should have control in choosing the overlay used to send their traffic, and should be able to leverage the overlay functions despite using overlay-unaware (legacy) applications.
- Factor out common functions: Instead of relying on the security provided by overlays, the architecture should provide basic security features such as host authentication and encryption.

3.2 Overlay Convergence Layer

OCALA interposes a layer, called the *overlay convergence (OC) layer*, between the transport and the network layers (see Figure 1). The OC layer replaces the IP layer in the Internet's stack, and consists of two sub-layers: an overlay independent (OC-I) sub-layer, and an overlay dependent (OC-D) sub-layer.

The main functions of the OC-I sub-layer are to present a consistent IP-like interface to legacy applications and to multiplex/demultiplex traffic between applications and various overlays. In addition, the OC-I sub-layer provides common functions, such as authentication and encryption, that are useful across overlays.

The OC-D sub-layer consists of modules for various overlays, which are responsible for setting up overlay-specific state and for sending/receiving packets to/from the particular overlay. For example, the i3 OC-D module maintains private triggers at both end-points, while the OverQoS module performs resource reservation. Note that IP can be viewed as a "default" overlay module.

Figure 2 shows an example in which three applications on host A open connections over IP and two overlays: a web browser (Firefox) uses IP to connect to a CNN server, a chat client (IRC) uses i3 to preserve its anonymity, and ssh uses RON for improved resilience. The design also enables hosts in different overlays to communicate with each other. Figure 3 shows how two hosts on different overlays can communicate using a gateway host (B) that is connected to both overlays.

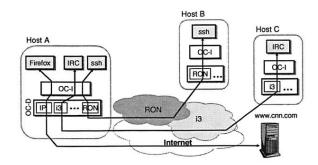


Figure 2: Three applications on host (A) which establish connections via IP and two overlays: RON and i3.

We refer to the end-to-end communication channel between two end-hosts at the OC-I sub-layer as a *path*, and to the communication channel between two end-hosts at the OC-D sub-layer as a *tunnel*. In Figure 3 the path between the two end-hosts is (A, B, C), and consists of two tunnels, (A, B) and (B, C).

3.3 Layering in OCALA: Discussion

The services implemented by the OC-I and OC-D sublayers on the data plane are analogous to the services provided by the network and data-link layers in the OSI protocol stack respectively. Like the data-link layer which provides communication between any two nodes in the same link layer domain, OC-D provides communication between any two nodes in the same overlay. Similarly, like the network layer which provides communication across different link layer domains, the OC-I sub-layer provides communication across different overlays.

However, OCALA does not enforce strict layering within its sub-layers. Unlike traditional layering, where a layer uses only the services provided by the layer below, OCALA allows legacy applications to access the services provided by the OC-D sub-layer, by passing overlay-specific names or IDs to OC-D through the OC-I sub-layer. These names are resolved at the OC-D sub-layer, and their semantics is opaque to the OC-I sub-layer. This allows us to achieve the main goal of OCALA—enable legacy applications to take advantage of the functions provided by overlays—while keeping the OC-I sub-layer agnostic of the overlays.

4 Detailed Architecture

We present a goal-driven description of OCALA, by showing how our design achieves the four goals we laid out in § 3.1. Achieving these design goals is challenging as they have conflicting requirements. For instance, on one hand, we want to expose the rich functionality provided by overlays to users, while on the other, we have to preserve the narrow IP interface exposed to the legacy applications. Our design aims to find a sweet spot in achieving these opposing goals.

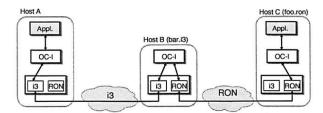


Figure 3: Bridging multiple overlays.

4.1 Goal 1: Achieving Transparency

Our main goal is to ensure that legacy applications are oblivious to the existence of overlays. Ideally, applications should work without any changes or reconfiguration when IP is replaced by the OC layer. Our design is fundamentally constrained by how a legacy application interacts with the external world. Most legacy applications make a DNS request, and then send/receive IP packets to/from the IP address returned in the DNS reply. Thus, legacy applications identify Internet hosts using names and IP addresses, where names are resolved using DNS to IP addresses.

We now describe and justify the following design decisions regarding names and IP addresses exposed to the legacy application:

- Overlay hosts are identified primarily using names.
 These names are resolved using overlay-specific resolution protocols. Each overlay can implement its resolution protocol, which may differ from a DNS lookup.
- The IP address returned to the application by the resolution protocol has only local meaning. This address serves as an OC-I handle to retrieve state corresponding to the remote host. Similarly, a tunnel descriptor is used by the OC-D sub-layer to maintain hop-by-hop state, and a path descriptor is used at the OC-I sub-layer to maintain end-to-end state.

4.1.1 Overlay Names

Users can exercise control over the overlay used for delivering their traffic by using: (a) fields in the IP headers, *e.g.*, IP addresses, port numbers, or (b) DNS-like names.

In the first approach, a user can specify rules on how packets should be processed using fields in the IP header. For example, the user can specify that packets sent to address 64.236.24.4 and port 80 should be forwarded through RON, while packets sent to 207.188.7.x should be forwarded through OverQoS.

In the second approach, users can encode the overlay to be used in the DNS names. We refer to the unique name associated with each overlay host as its *overlay name*. An overlay name is of the form *foo.ov*, where *ov* specifies the overlay, and *foo* is a name unique to

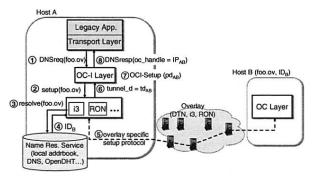


Figure 4: Path setup protocol.

that overlay. On receiving a DNS request for an overlay name, the OC layer sets up state which allows it to intercept and forward all the subsequent packets from the application to host *foo.ov* through overlay *ov*.

The main advantage in relying solely on the information in the IP headers is that it works with *all* Internet applications, since at the very least, an application sends and receives IP packets. On the other hand, using overlay names has several advantages. First, overlay names can be used to identify hosts (for example, NATed hosts) without routable IP addresses. This property is fundamental to overlays that bridge multiple address spaces [3, 22]. Second, names are human-readable and hence easier to use. Third, the user does not need to know the IP address of the destination in *advance*, which is not feasible in some cases. Indeed, when an overlay provides support for content replication, the IP address of the server that ultimately serves the content may not be known to the users.

Our implementation chooses DNS-like names as the primary method for overlay selection. For supporting applications that do not make DNS requests, we also support the use of IP header fields for overlay selection.

4.1.2 Overlay-specific Resolution

Our second design decision is to resolve overlay names using overlay-specific mechanisms. A name of the form foo.ov, is resolved by the OC-D module for overlay ov. This design choice has two main advantages over DNS-based resolution. First, this allows multiple namespaces to co-exist with each other and with the DNS namespace, thus enabling a fully extensible namespace. Each overlay is allowed to implement its name allocation and resolution, without requiring a global infrastructure. Second, this allows OCALA to support network architectures that do not assume global IP address allocation. Examples include MetaNet [41] and IPNL [8] wherein names are the only way to refer to hosts. Other examples include architectures that leverage name resolution to implement different functions (e.g., DoA [39]).

In the remainder of the section, we describe how the

control plane and data plane operations of OCALA transparently set up an end-to-end path and tunnel the legacy applications' data across the overlay.

4.1.3 Control Plane: End-to-End Path Setup

A new connection setup is triggered by the receipt of a DNS request for a previously unseen overlay destination or the receipt of the first data packet of a connection configured to use a particular overlay. The final result of these operations is establishing an end-to-end *path* at the OC-I sub-layer and setting up the state required to handle the application's traffic. While a path could consist of several *tunnels* at the OC-D sub-layer, in this section we consider a single-tunnel path. We generalize the description to multi-tunnel paths in § 4.2.

Consider a legacy application on host A that wants to communicate with a remote legacy application at host B, called foo.ov (see Figure 4). The application first issues a DNS request for foo.ov, which is intercepted by the OC-I sub-layer. On receiving such a request, the OC-I sub-layer associates a globally unique path descriptor, pd_{AB} , and remembers the mapping between the name and the descriptor $(foo.ov \rightarrow pd_{AB})$ in order to service future requests for foo.ov. We minimize collisions by randomly choosing the path descriptor from a 128-bit number space.

The OC-I sub-layer then invokes the corresponding module in the OC-D sub-layer to setup a tunnel to foo.ov through overlay ov. In turn, the OC-D sub-layer invokes a resolution service to obtain the overlay ID (ID_B) of foo.ov. Examples of resolution services are DNS (used in RON), OpenDHT [16] (used in DOA), and implicit name-to-identifier hashing (used in i3). After the OC-D sub-layer resolves the name, it instantiates the necessary state for communicating with foo.ov, and returns a pointer to this state, the $tunnel\ descriptor$, td_{AB} , to OC-I. For example, in i3, the setup phase involves negotiating a pair of private triggers with the remote end-host, and instantiating the mapping state between foo.ov and the private trigger IDs.

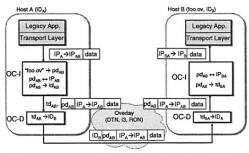


Figure 5: Forwarding a data packet from host A (with IP address IP_A) to B (with IP address IP_B). The mappings used to modify the packet are in bold.

4.1.4 Data Plane: Packet Forwarding

The application at host A addresses packets destined to foo.ov to IP_{AB} , the OC handle returned by the OC-I sublayer (see Figure 5). The OC-I sub-layer retrieves the state associated with this handle, and appends the path descriptor, pd_{AB} , to the packet, before handing it off to the OC-D sub-layer to be sent over tunnel td_{AB} . The OC-D sub-layer, using its tunnel state, sends the packets to foo.ov using the overlay ID, ID_B . At the destination, the packet is handed to the OC-I sub-layer, which uses the path descriptor in the header to demultiplex the packet. Before sending the packet to the application, the OC-I sub-layer rewrites the source address to IP_{BA} , the OC handle associated with the A to B path at B. The destination address is rewritten to the local IP address at B.

As evident from this description, the constraint imposed by supporting unmodified applications leaves us with little choice but to overload the semantics of application-level names and IP addresses. We discuss the limitations of overloading names and addresses on transparency in § 4.5.

4.2 Goal 2: Bridging Multiple Overlays

When multiple overlays are deployed, a potential undesirable side-effect is that hosts in different overlays may not be able to reach one another. For example, i3 allows NATed hosts to act as servers, but such servers will be unreachable through RON. Even in the Internet, hosts in different IP address spaces cannot communicate with one another [22]. Moreover, it is likely that some of the Internet hosts will not participate in overlays.

Our architecture addresses these problems by allowing remote resolution of names, a mechanism borrowed from other architectural proposals such as DOA [39]. When a host belonging to overlay ov1 resolves an overlay name foo.ov2, the OC-I sub-layer resolves the name by forwarding the request to a gateway which participates in the overlay ov2. We provide interoperability between overlay and legacy hosts by designing special OC-D modules that send and receive IP traffic to and from legacy hosts.

When performing remote resolution, path descriptors are used as state handles across intermediate hops (such as gateways). The tunnel descriptor is a handle passed between the OC-I and the OC-D sub-layers at the *same* host; the path descriptor is used as a handle between the OC-I sub-layers at *different* hosts. Thus, intermediate hops can use the path descriptor to retrieve state required to relay the packet further. Further, decoupling path and tunnel descriptors allows different paths to share the same tunnel. For example, paths (A, B, C) and (A, B, D) can share the tunnel (A, B).

We now describe our mechanisms to bridge different overlays in more detail.

4.2.1 Overlay Gateways

Consider a host A that uses the i3 overlay wishing to contact a host C in the RON overlay (See Figure 3). To enable this communication, we deploy a host (gateway) B that resides on both i3 and RON, and runs the OC-D modules for both overlays. Host A then sets up a two-hop path to C by using the gateway as an intermediate hop. For a multi-hop path, the setup protocol creates tunnels between consecutive hops and sets up the routing state at the OC-I sub-layer of the intermediate hop to create an end to end path. We now give the details of the protocol.

Assume that the overlay name of host C is *foo.ron*. Configuration files at host A (described in § 4.3) indicate that connections to *foo.ron* should go through a gateway B in *i*3 with the name *bar.i3*. To communicate with host C, an application at host A issues a DNS request for *foo.ron*. The OC-I sub-layer, upon intercepting this request, instructs the *i*3 OC-D module to set up a tunnel to *bar.i3*. This operation is identical to the tunnel setup in § 4.1.3. Once this tunnel is setup, the OC-I at A asks its peer at B to set up the rest of the path to the destination C recursively.

At the end of the setup protocol, an end-to-end path is established from A to C with the unique path descriptor pd. A common path descriptor helps identify a path so that any path breakages can be dynamically detected and quickly repaired. Our gateway, as in the case of a NAT, maintains per-path state.

4.2.2 Legacy Gateways

Legacy gateways are similar to overlay gateways except that one of the tunnels is over IP to a legacy host that does not participate in any overlay natively and does not run the OC-I sub-layer. Thus, overlay functionality, such as improved routing, will be available only on the tunnel established over the overlay (between an overlay host and the gateway). There are two types of legacy gateways:

Legacy server gateway. The legacy server (LS) gateway allows an overlay-enabled client to contact a legacy server (see Figure 6(a)). Functionally, the LS gateway

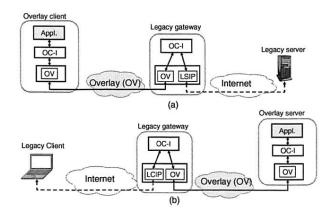


Figure 6: (a) An overlay client connecting to a legacy server. (b) A legacy client connecting to an overlay server.

runs an OC-I sub-layer over an OC-D module (say i3) and a special OC-D module called LegacyServerIP (or LSIP). The setup protocol is similar to that for an overlay gateway. Consider a overlay host connecting to cnn.com through the LS gateway. The OC-I sub-layer at the LS gateway forwards such setup requests to the LSIP module. The LSIP module now behaves like a NAT box with respect to the server. It first resolves the name cnn.com using DNS and allocates a local port for this tunnel. Packets sent to the server are rewritten by changing the source address to that of the LS gateway, and altering the source port to be the allocated local port. The local port is then used to multiplex incoming packets, which are then sent to the OC-I sub-layer with the appropriate handle.

Legacy client gateway. The legacy client (LC) gateway enables overlay servers to offer their services to legacy clients; legacy clients are not overlay enabled, nor do they run the OC-I sub-layer (see Figure 6(b)). The LC gateway runs the OC-I sub-layer over an OC-D module (say i3) and a special OC-D module called LegacyClientIP (or LCIP). In addition, the client is configured to use the LC gateway as its DNS server. The LCIP module intercepts DNS queries from the client, and dispatches them to the OC-I sub-layer which initiates a tunnel over the overlay. The LCIP module then sends a DNS reply with an Internet routable address to the client, captures packets sent by the legacy client to that address, and sends them over the overlay. Any client can now contact the machine foo.i3 from any machine provided that its DNS server is set to the address of the LC gateway. The design of our LC gateway is similar to that of AVES [22]. The fact that the addresses returned by the gateway should be routable considerably limits the number of clients that can connect simultaneously [22]. HTTP traffic does not suffer from this limitation since gateways can use DNS names in the HTTP requests for demultiplexing.

Figure 7: Configuration snippet indicating that ssh traffic or connections to all DNS names ending in .ron should go over an instance of RON running on PlanetLab, using the minimum latency metric.

4.3 Goal 3: Exposing Overlay Functionality

Different new architectures and overlays provide different functions. Users should be able to choose the overlay best suited for a particular application. The overlay selected might allow further customization of the functions it offers. For example, RON allows users to choose the metric based on which the paths are optimized, OverQoS allows users to specify QoS parameters, and architectures like i3 and DOA allow users to explicitly interpose middleboxes on the path. For flexibility, users should be able to customize their preferences for each tunnel along a path. Preferences include both overlay-specific (e.g., use latency optimized paths for RON) and overlay-independent options (e.g., identity of gateways, end-to-end authentication).

Given the limited options available to a legacy application for communicating its preferences to the OC layer, our initial design was to encode the user preferences in the DNS name. For example, a DNS name foo.delay50ms.overqos was used to identify a connection to the host with name foo using a path of less than 50 ms delay in OverQoS. However, overloading DNS names to include preferences had multiple disadvantages, from highly restrictive syntax to being plain cumbersome to utilize. Although this approach is implemented in OCALA, we do not use it.

Instead, we opted for expressing user preferences using XML configuration files. On receiving a setup request for an overlay name, the OC-D sub-layer reads the preferences associated with the name (if any) from the configuration file, before proceeding with the setup operation. A snippet from a configuration file is shown in Figure 7. Though directly manipulating the configuration files offers great flexibility, we expect users to rely on our graphical user interface described in § 7.

4.3.1 Support For Middleboxes

OCALA also allows users to customize their data path by redirecting traffic through specific middleboxes using the configuration files. Several new network architectures [32, 39] provide support for such middleboxes,

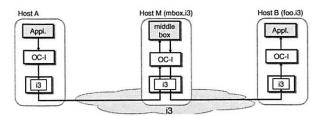


Figure 8: Interfacing a middlebox.

by allowing both the sender and the receiver to *explicitly* insert middleboxes on the data path. OCALA's support for middleboxes is similar to that for gateways. Consider the case of a sender-imposed middlebox where a host A wishes to contact a host B through a middlebox M (see Figure 8). The only difference from the operation of a gateway is that the middlebox module running at M should be allowed to perform arbitrary transformations on the data sent by one end-point before forwarding it to the other. In OCALA, the middlebox module implements a single function call that is used by the OCI sub-layer to pass packets to it. A configuration file at M specifies the middlebox operations to be applied to connections traversing the middlebox. The protocol for receiver-imposed middleboxes is similar.

4.4 Goal 4: Factoring Out Common Functions

A second-order goal aimed at reducing the effort of overlay developers, is to leverage the OC-I sub-layer to implement generic functions, such as security and data compression, that can be used by different overlays.

Security and authentication of data connections are important requirements for many overlays, especially in cases where flat names are employed. OCALA incorporates basic security mechanisms at the OC-I sublayer. In particular, the OC-I sub-layer offers encryption and authentication, both of which operate agnostic of the overlay used for the traffic. The OC-I sub-layer's authentication mechanism is based on human-readable names and relies on the existence of a certification and name-allocation authority from which users can obtain certificates associating their overlay name to their public key.² OCALA's protocol for securely communicating with a host known by its name alone is very similar to the Secure Sockets Layer protocol (SSL) [10] which relies on certificate authorities like VeriSign. We designed our own custom protocol rather than reusing SSL since in general middleboxes need to operate on unencrypted data, which is not possible under the existing end-to-end model of SSL.

²We do not know of any mechanism for eliminating the centralized authority for a human-readable and secure naming scheme. It is easy to extend our model to hierarchical nameallocation schemes.

4.5 Limitations

The primary goal of our design is to achieve transparency for legacy applications while providing complete access to overlay functions. We review how well our design meets this goal.

4.5.1 Access to Overlay Functions

While the OC layer enables legacy applications to take advantage of most overlay functions such as mobility, anycast, QoS, route optimizations and middleboxes, there are two important limitations. First, the fact that OCALA is positioned below the transport layer makes it hard, if not impossible, for legacy applications to take advantage of overlay networks that provide transport- or application-layer functionalities (e.g., multi-path congestion control, or data storage [17]). Second, the current instantiations of OCALA support only unicast legacy applications; it provides no support for legacy applications using IP multicast; we are currently designing a multicast abstraction at the OC-I sub-layer.

4.5.2 Transparency

The OC-I sub-layer overloads IP addresses in ways that might break assumptions made by some applications. In contrast to usual IP semantics, the scope of addresses returned by the OC-I sub-layer to applications is local. Firstly, the use of local scope addresses implies that addresses returned to legacy applications will not be valid at other hosts. In our experience, this does not break several common applications like ssh, Internet Explorer, remote desktop, and ftp servers. However, peer-to-peer applications and SIP may not work under OCALA (unless all hosts run OCALA). Secondly, applications like ftp that encode addresses in data packets will potentially not work since the OC-I sub-layer performs IP header rewriting before delivering packets to the application. Our implementation avoids address rewriting to some extent by negotiating the local addresses at the OC-I sub-layer, a technique borrowed from [42]. However, for legacy gateways, address rewriting cannot be avoided.

Local-scope addresses have been used before in several contexts and their limitations and workarounds are well-known [42]. In supporting overlays where hosts may not have routable IP addresses, we are left with little choice but to work around the limitations of local-scope addresses using mechanisms like address negotiation.

5 The Overlay Dependent Layer

The overlay dependent sub-layer implements the functions specific to an overlay. We first present the interface that is exported by an OC-D module to the OC-I sub-layer. We then describe the working of the OC-D modules for two overlays, i3 [32] and RON [1], which we developed in-house. This description serves not only as

a validation of our architecture but also as a blueprint for implementing OC-D modules for other overlays.

5.1 OC-D Module API

Table 1 shows the basic API functions that every OC-D module needs to implement and expose to the OC-I sublayer. For simplicity of exposition, we omit error-related functions here.

Function calls	s: OC-I \rightarrow OC-D
setup(name,pref, path_d)	setup path to host <i>name</i> using preferences <i>pref</i>
close(tunnel_d)	close tunnel
send(tunnel_d, IP_pkt)	send IP packet via tunnel
Callbacks:	OC-D → OC-I
setup_done(path_d, tunnel_d)	callback invoked when tunnel (tunnel_d) was established
recv(path_d, IP_pkt)	receive IP packet from tunnel

Table 1: OC-D Module API.

The basic API consists of three functions and two callbacks. The setup function sets up a tunnel between the local host and a remote host according to the user's preferences. The user preferences *pref* and the overlay name of the remote host *name* are passed in the setup call. The *path_d* field represents the path descriptor at the OC-I sub-layer and is used by the OC-D sub-layer in the *setup_done* callback. Once the OC-D sub-layer creates the tunnel it returns the tunnel descriptor (*tunnel_d*) to the OC-I sub-layer using callback *setup_done*. The close function call is invoked by the OC-I sub-layer to close the specified tunnel. This function is usually called when a path's state at the OC-I sub-layer expires. We discuss the timeout values for this state in the implementation section (§ 7.1).

The send function call, invoked by the OC-I sublayer, includes a handle to the OC-D's state for that tunnel (i.e. the tunnel descriptor) and the packet itself. The recv call, is invoked by an OC-D module to the OC-I sub-layer, upon receiving a packet from the overlay.

5.2 The RON Module

RON [1] aims to improve the resilience of the Internet by using alternate routes in the overlay. RON offers an interface similar to IP, and not surprisingly, it requires very little effort to implement the OC-D module for RON. RON uses IP addresses and DNS names as overlay IDs and overlay names, respectively.

When the OC-I sub-layer asks the RON module to setup a connection to a RON host (identified by a name such as *foo.com.ron*), this name is resolved using the DNS infrastructure to obtain an IP address. The RON module then sets up state associating the preferences and the destination IP address with the tunnel and passes its

handle to the OC-I sub-layer. Data plane operations involve simple encapsulation and decapsulation.

5.3 The *i*3 Module

i3 [32] is a network architecture that uses a rendezvous-based communication abstraction to support services like mobility, multicast, anycast and service composition. We now describe how the i3 module works when host A contacts host B over i3.

On receiving the setup request for B.i3 from the OC-I sub-layer, the i3 OC-D module at A first resolves the name to a 256-bit i3 identifier by using implicit mapping: the identifier of a host is derived by simply hashing its name. The identifier obtained by hashing B.i3 corresponds to B's public trigger identifier id_B . Thus, i3 does not require any resolution infrastructure.

After the name is resolved, the i3 module at A initiates private trigger negotiation by contacting host B through its public trigger $[id_B|B]$. Both hosts exchange a pair of private triggers $[id_{AB}|A]$ and $[id_{BA}|B]$, respectively, after which they communicate exclusively through these triggers: host A sends packets to host B using ID id_{BA} , and host B sends packets to A using ID id_{AB} . Once the control protocol sets up the required state, the i3 module sends packets captured by the OC-I sub-layer by encapsulating the payload with i3 headers that include the private triggers identifying the flow.

The i3 OC-D module allows receiver-imposed middleboxes by using i3's stack of IDs. An i3 host B that wishes to impose the middlebox M on all hosts contacting it, inserts a public trigger of the form $[id_B|(id_M,B)]$. When a client A sends a trigger negotiation request via the ID id_B , i3 delivers it to M along with the stack (id_M,B) . The i3 OC-D module thus obtains the identity of the next hop and automatically proceeds to set up the tunnel to B through its OC-I sub-layer.

6 Applications

Legacy applications benefit from OCALA in two different ways. Firstly, OCALA enables applications to leverage the new functionality offered by overlays. Secondly, the OC-I sub-layer of OCALA allows a path to traverse multiple overlays thus composing their functionalities. We now describe some applications that demonstrate these two types of benefits.

6.1 Functions Enabled by Overlays

NAT Traversal: Since i3 enables access to machines behind NATs, a user can run legacy servers behind NATs by using the i3 OC-D module. In addition to allowing external hosts to contact these servers, OCALA also enables users to securely access their home machines from anywhere by using the human-readable i3 name of the home machine. When persuading users to deploy OCALA, we

found NAT traversal to be a very attractive feature from the users' perspective.

Receiver Imposed Middleboxes: i3 enables hosts to redirect all incoming traffic to a middle-box which can be located anywhere in the network. We used this ability to force all traffic sent to a legacy web server to pass through an intrusion detection middlebox which was not located on the physical path to the server. We used the popular Bro [24] intrusion detection program in our implementation by writing a 200-line middlebox shim layer through which the OC-I sub-layer relays packets that are to be analyzed by Bro.

Observe that Bro is itself a legacy application, and thus packets sent to Bro should have valid IP headers. For this reason, the shim layer assigns virtual addresses to both end points and rewrites the IP headers appropriately, before sending the packets to Bro. To Bro, communication between the remote hosts looks like a conversation between two virtual hosts, and it can perform stateful analysis (e.g., TCP analysis by matching the data packets of a TCP connection with the corresponding acknowledgments). Since Bro sees only virtual addresses, it cannot perform certain analysis like address-scan detection that looks for several unsuccessful connection attempts to hosts within the same network.

Secure Mobility: HIP enables hosts to securely communicate with each other even when the hosts are mobile. We leverage this functionality of HIP to support ssh connections that remain alive even when one of the hosts changes its IP address.

6.2 Functions Enabled by the OC-I Sub-layer

Secure Intranet Access: We implemented a flexible and secure version of Virtual Private Networks (VPNs) [37] by using the OC-I sub-layer to contact legacy hosts using an overlay. A legacy server gateway runs inside the organization and hence has unrestricted access to all intranet hosts. To access intranet machines, external hosts relay packets through the legacy gateway. Authentication and encryption are important requirements in this scenario, and we leverage the OC-I sub-layer's security mechanisms. Any routing overlay, including vanilla IP, can be used for communicating between the user's machine and the legacy gateway. The main advantage of our system over VPN-based systems is that a client can access multiple intranets at the same time even if all intranets use the same address range. Users specify their preference through the configuration file-e.g., all connections to *.company1.com should go through the gateway1 of company 1 while connections to *.company2.com should use the gateway of company 2. Another distinguishing feature of our system is that a client is not assigned an IP address from the intranet address space. This improves the security of our system by making it difficult for a client infected by a scanning worm to directly attack other hosts within the intranet.

Overlay Composition: Overlay composition allows an application to explicitly stitch together different network overlays. Apart from enabling inter-operability, stitching allows a user to merge the functions offered by different overlays. For example, a user who connects to the Internet through a wireless hop, may use i3 for uninterrupted communication while switching between various wireless networks. In addition, the user may wish to optimize wide-area performance using RON. We achieve this by using i3 to connect to a close-by i3-to-RON gateway, which will then relay packets over a RON-optimized path over the wide-area.

7 Implementation

We have implemented the OC-I sub-layer as a user-level proxy. Although OCALA inserts a new layer into the network protocol stack, our implementation avoids modifications to the operating system by using the tun [38,40] packet capture device. The OC-I sub-layer reads from the tun device to capture packets sent by legacy applications and writes to it to send back replies.

The OCALA proxy and the configuration GUI consist of approximately 30,000 source lines of code (SLOC) in C++ and 6,000 SLOC in Java respectively. The software, which currently works on Linux, Windows XP/2000 and Mac OS X, is available at http://ocala.cs.berkeley.edu.

We have implemented OC-D modules for RON and i3 using source code available from their project websites. The HIP, DOA and OverDoSe OC-D modules were independently implemented by external research groups. An OC-D module is a C++ class implementing the API of the OC-D base class, compiled into .so, .dll and .dylib dynamically loaded libraries in Linux, Windows and OS X respectively. OC-D modules are dynamically loaded and plugged into the proxy based on user configuration. In its simplest form, an OC-D module translates between OC-I API calls and overlay-specific functions. In our experience, implementing an OC-D module is a simple task requiring less than 200 lines of code. We only count the code used to interface the OC-D to the OC-I, and not the code used to implement overlayspecific functionality.

Users control the proxy and express their preferences (e.g., ssh traffic should go over RON, Internet Relay Chat should use i3) through a set of XML configuration files. We have implemented a graphical user interface that enables users to set their preferences without manually editing XML files. The GUI has a modular design which enables developers to plug in components which expose overlay-specific configuration options to users.

Our implementation requires administrative privileges for using the tun device and forces all users on the same machine to share the same configuration. These limitations can be avoided by a dynamic library-based implementation.

In the remainder of the section, we describe the implementation of the control plane, data plane and gateway operations in detail.

7.1 Control Plane: State Maintenance

Control plane setup begins when the OC-I sub-layer intercepts a DNS request for a previously unseen destination. The OC-I sub-layer initializes state, such as path descriptors, and communicates with its peer OC-I sublayer(s) to set up the end-to-end path requested by the application. If the application requires, the same localscope address is negotiated at both end points. If security is enabled, the protocol authenticates the nodes on the path and establishes 256-bit symmetric keys for each tunnel. These protocols are piggybacked on top of path setup to reduce latency. After setup completion, the OC-I sublayer sends the DNS reply containing the local-scope address to the application. The local-scope addresses are allocated from the unused address range 1.0.0.0/8. To prevent caching, the Time To Live (TTL) of the DNS reply is set to zero. The state associated with a path times out and is removed if no data packets are sent or received on that path for 7200s. This large timeout period was chosen to deal with applications like Internet Explorer which we found to cache DNS replies beyond their specified TTL. When the path is alive, periodic keep-alive messages are exchanged between the sender and the receiver to quickly detect and repair any breaks in the end-to-end path.

7.2 Data Plane: Packet Forwarding

Packets sent by the application are addressed to the local-scope addresses returned by the OC-I sub-layer after path setup. The OC-I sub-layer intercepts packets sent to local-scope addresses, as well as packets which match patterns (based on addresses and ports) that are explicitly specified in the configuration file. Depending on user preference, the OC-I sub-layer may compress or encrypt the packet before dispatching to the OC-D sub-layer. The headers added by the OC-I and OC-D sub-layers may lead to packet fragmentation; fragmentation can be avoided if applications perform end-to-end MTU discovery. Rewriting of addresses at the destination occurs only if local-scope address negotiation between the end points had failed during path setup.

7.3 Legacy Gateways

Our LSIP implementation includes packet-rewriting support for several applications such as FTP, H.323, PPTP and SNMP. The legacy server gateway does not support ICMP since there is no information in an ICMP packet

(such as port numbers) to permit multiplexing of a single IP address among multiple hosts. The LCIP implementation is very similar to AVES [22], and a legacy client can connect to a name of the form *foo.i3.ocalaproxy.net* in order to communicate to the webserver at *foo.i3*.

8 Evaluation

The purpose of our evaluation is to demonstrate that the overhead of packet capturing and tunneling in our implementation is not large. The real benefit of our architecture and implementation should be evaluated by the applications it enables, and eventually, the user acceptance it gains. We first micro-benchmark the data and control paths of the proxy, and then present local-area and wide-area experiments.

8.1 Micro-benchmarks

Micro-benchmarks were conducted on a 2.4 GHz Pentium IV PC with 512 MB RAM running Linux 2.6.9. An in-house tool that sends packets at a specified rate played the role of a legacy client. Both the proxy and the tool were instrumented to record the timestamps at relevant checkpoints. Each timing statistic reported here is a median of 100 runs.

Data Path Overhead. In comparison to a legacy application running over the host IP stack, the proxy adds two memory copies: from kernel to user space and back, both while sending and receiving packets. Table 2 reports the send and receive times of a single packet of size 1200 bytes 3 for i3 and RON 4 . The total send and receive times are split into three phases: (a) time to move a packet between the application and the proxy (using tun), (b) overhead at OC-I sub-layer, and (c) overhead at OC-D sub-layer.

	Send		Receive	
(µs)	i3	RON	i3	RON
OC-I	19	18	8	6
OC-D	20	28	44	36
tun	24	24	16	15

Table 2: Split-up of per-packet overhead for send and receive.

As expected, the processing time of the OC-I sub-layer is independent of whether we use i3 or RON. The percentage of time spent in the OC-I sub-layer is not large— 28% for send and 11% for receive (on enabling OC-I features like encryption, the overhead rises to more than 67%). The remaining overhead is almost equally split between OC-D processing and transferring the packet from

the application to the proxy. Although the i3 and RON OC-D modules are very different, the processing times associated with them are similar. A dynamic library implementation can reduce the overhead of packet transfer between the application and proxy, by avoiding extra packet copying. The total processing time indicates that the proxy can sustain a throughput of about 15000 packets per second (for 1200-byte packets).

Control Path Overhead. Path setup is triggered when a DNS request made by an application is captured. If a path for the requested name was previously set up, the proxy immediately answers the DNS query with a small processing overhead of 15 μ s. Otherwise, it performs additional operations to set up the path and hence takes longer (169 μ s) to respond to the application.

8.2 LAN Experiments

In order to study the effect of the proxy overhead on end-to-end behavior, we measured (Table 3) the latency and TCP throughput between two clients communicating over i3, i3-shortcut⁵, RON and plain IP, within the same LAN. In a LAN environment, the overhead of the proxy can be localized without wide-area artifacts affecting the measurements.

	i3	i3-shortcut	RON	IP
Latency (ms)	1.42	0.788	0.762	0.488
Tput. (KBps)	9589	10504	10022	11749

Table 3: LAN experiments for latency and throughput.

Latencies under i3-shortcut and RON are a few hundred microseconds larger than IP latency. Since LAN latencies are themselves very small, even a single intermediate server on the data path causes significant relative increase in latency for i3. The throughput results (average over 10 measurements) indicate that the performance hit due to proxy and overlay overheads is only about 10%. The throughput and latency of RON is not better than IP since in this simple experimental setup, all RON and IP packets traverse the same LAN. Since the i3 servers were also located on the same LAN, relaying packets through i3 did not cause significant throughput degradation.

8.3 Wide-area Experiments

We measured OCALA's performance over i3, i3-shortcut, RON and plain IP in the wide area. We also measured the performance when traffic traversed i3-RON, i3-IP and RON-IP gateways. Difficulty in obtaining hosts with root privileges limited our experiments to just three machines at Berkeley, Stanford and Boston, which we refer to as A, B and C respectively. Latency

³We used this packet size in order to avoid fragmentation due to addition of headers.

⁴We benchmark only the two OC-D modules that we implemented; the others were implemented external to our research group.

⁵Shortcut is an *i*3 optimization that eliminates the inefficiency of relaying packets through intermediate *i*3 servers.

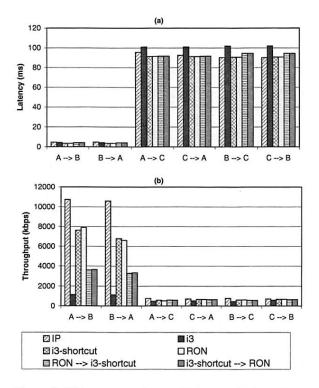


Figure 9: Wide-area experiments: (a) latency (b) throughput.

was measured using ping, and throughput was measured using ttcp. i3 and RON networks were deployed on PlanetLab. The i3 OC-D module on the end-host used the closest i3 server, while the end-host itself joined the RON network using its RON OC-D module.

We first consider the latency and throughput results for the single network scenario. Figure 9(a) shows that latencies for i3-shortcut and IP are nearly equal. This is not surprising, as in both cases, packets follow the direct IP path between the end-points. Although we configured RON to choose latency-optimized paths, we observed no significant improvements in latency compared to the direct IP path. Due to the limited size of our experiment, the path with the best latency was always the direct IP path. Plain i3 incurs larger latency as packets are forwarded via an intermediate i3 server. In a few experiments, IP incurred a higher latency than i3 and RON; we attribute this to UDP packets getting preferential treatment over ICMP ping packets (note that packets are encapsulated in UDP when i3 or RON are used). We confirmed this by measuring latencies using the UDP Echo [27] protocol, wherever permitted by firewalls.

Throughput measurements in Figure 9(b) indicate that i3 performs much worse than the direct IP path. Throughput over i3 and RON vary between 62% and 95% of the direct IP throughput. We attribute this performance degradation to the extra headers added to each packet and the proxy processing overheads. We further suspect that TCP packets are getting preferential treat-

ment over UDP in the wide area.

i3-RON bridge. We measured throughput and latency between each pair of machines, with one of the machines in the pair connected only to i3 while the other was connected only to RON. A second machine (D) at Berkeley acted as an i3-RON gateway. As shown by Figure 9(a), the increase in latency for the bridged path over the direct IP path is small. However, the presence of the i3-RON gateway on the path resulted in lower throughput. The adverse effect of bridging is dominant when nodes are very close to each other. For example, throughput between the Berkeley and Stanford nodes under bridging is approximately one-third of the direct IP path, while for distant nodes (Berkeley-Boston, Stanford-Boston), the throughput drop is less than 20%.

Legacy Server Proxy. We ran i3-IP and RON-IP legacy server proxies on machine D. The proxies at A, B and C were configured to relay connections to mozilla mirrors (http://www.mozilla.org) through the server proxies, with the first hop using i3 or RON. The server proxies connect to the mozilla mirror on behalf of A, B or C. To measure throughput, we downloaded 10 different files from 10 different mozilla mirrors. The average throughput while using the i3-IP and RON-IP gateways was within 85% of the throughput obtained while directly downloading the same set of files.

The main reasons for reduced throughput in both wide area and LAN experiments are the overheads due to extra headers and relaying through intermediate hops (for bridging). These are inherent limitations of tunneling.

8.4 Number of Simultaneous Connections

OCALA proxy can handle a large number of simultaneous connections. Due to the difficulty in procuring machines with root privileges, our experiment is limited to the legacy server proxy scenario where one of the end-points of a connection is a legacy host not running OCALA. We used an 8 machine cluster (Intel Xeon 3.06 GHz, 2GB RAM each) for running the OCALA proxies. Machine 1 of the cluster ran a legacy server proxy. Machines 2 to 8 ran normal client proxies configured to use the legacy server proxy at 1 for relaying all legacy traffic. From each of machines 2 to 8, we accessed 175 legacy websites in parallel. The legacy server proxy on 1 was able to service over 1000 simultaneous connections with 26% CPU utilization and 0.4% memory utilization.

8.5 Path Robustness

OCALA's periodic *keep-alive* messages enable broken paths to be quickly detected and repaired. If no *keep-alive* responses are received for 5s, OCALA invokes path re-establishment. This mechanism retries every 10s till a path is set up. Again, due to difficulty in obtaining machines with root privileges, we measured the time taken

to detect and repair a broken path between just two machines (at Berkeley and Boston). We experimented with paths consisting of one and two gateways. The average time to detect a path break was about 3s and the average time to repair the path was about 5s. These numbers agree with the reply wait timer and the path setup retry timer values in our implementation. (Due to difficulties in synchronizing the time across the different wide-area machines, we are unable to report results with finer time granularity.)

9 Discussion

In this section, we summarize our experiences with the OCALA deployment. We (and other groups) have used various versions of the proxy since March 2004. Over this time interval, the OCALA proxy has attracted interest from both overlay developers and endusers. Developers of various routing overlays and network architectures, such as Delay Tolerant Networks [6], OverQoS [35], Tetherless Computing [30], and QoS Middleware project [20], have expressed interest in leveraging the OCALA proxy for their own overlays.

The proxy has been used for supporting a variety of applications including *ssh*, *ftp*, web browsing, and virtual network computing (VNC) applications. Most end-users have typically used the proxy for accessing their home machines to get around NAT boxes and dynamic IP address allocation by their ISPs.

Based on our own experience and the feedback from other end-users and developers, we have learned a few lessons, some of which are obvious in retrospect. These lessons emphasize what is arguably the main benefit of OCALA: the ability to "open" the overlays to real users and real applications. The feedback received from such users has been invaluable in improving the OCALA design, and in some cases, the overlay design.

Efficiency matters. When using legacy applications, the users expect their applications to perform the "same" way no matter whether they run directly on top of IP or on top of an overlay. In particular, more often than not, we found the users unwilling to trade the performance for more functionality. This feedback led not only to proxy optimizations, but also to overlay optimizations. For example, the developers of *i*3 have added shortcuts to improve the end-to-end latency, and added the ability to share a private trigger among multiple tunnels to decrease the setup cost.

Security matters. Security was not part of our original design agenda. However, we found that the users expected at least the same level of security from the OCD name resolution mechanism as they get from today's DNS (where impersonation while possible, is not trivial). In the area of mobility, the users and developers argued

for even much stronger security guarantees such as authentication and encryption. In the end, this feedback led us to implement security in the OC-I sub-layer.

Usage is unexpected. Initially, we expected mobility to be the most popular application. However, this was not the case. Instead the users were more interested in using OCALA for such "mundane" tasks as accessing home machines behind NATs or firewalls, and getting around various connectivity constraints. In one instance, users leveraged the fact that the proxy communicates with *i*3 via UDP to browse the web through an access point that was configured to block TCP web traffic! The unexpected usage led us to provide better support for applications over NATs. In particular, we have implemented an OC-I handle negotiation mechanism that preserves the addresses in the IP headers. This allows us to support some applications that otherwise do not work over NATs (e.g., ftp).

10 Conclusion

Overlay networks have been the focus of much research in recent years due to their promise of introducing new functionality *without* changing the Internet infrastructure. Surprisingly little attention has been devoted to achieving the same desirable property at the end-host: provide access to new network architectures without any changes to legacy software such as operating systems, network applications, and middlebox applications.

Our work is a preliminary step in this direction and aims to improve the inter-operability between legacy applications and new network architectures, and between different network architectures. Currently, we (and others) are in the process of extending the OC-D sub-layer to support other overlay networks. Ultimately, we plan to enlarge our user base and gather more feedback to improve the proxy. As our experience showed, users often find unexpected uses to the system, which can push the design in new directions.

Acknowledgements

We thank David Andersen, Tom Anderson, Dennis Geels, Boon Thau Loo, Sridhar Machiraju, Sean Rhea, Mukund Seshadri, Scott Shenker, Arun Venkataramani, Mythili Vutukuru, and Michael Walfish for useful discussions and comments about the paper. We thank Stefan Goetz for his crucial contributions in porting the OCALA proxy to Windows and Mac OS X. We are grateful to Keith Sklower and Mike Howard for their help in deploying the OCALA legacy server and client proxies on a big scale.

References

 D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of SOSP*, 2001.

- [2] K. Argyraki and D. Cheriton. Loose Source Routing as a Mechanism for Traffic Policies. In *Proc. of FDNA*, 2004.
- [3] D. R. Cheriton and M. Gritter. TRIAD: A New Next Generation Internet Architecture, Mar. 2001. http:// www-dsg.stanford.edu/triad.
- [4] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *Proc. FDNA*, 2003.
- [5] S. E. Deering. Multicast routing in internetworks and extended lans. In *Proc. SIGCOMM*, 1988.
- [6] K. Fall. A delay tolerant network architecture for challenged internets. In *Proc. SIGCOMM*, 2003.
- [7] B. Ford. Unmanaged Internet Protocol: Taming the Edge Network Management Crisis. SIGCOMM CCR, 34(1):93–98, 2004.
- [8] P. Francis and R. Gummadi. IPNL: A NAT-Extended Internet Architecture. In *Proc. of SIGCOMM*, 2001.
- [9] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing Content Publication with Coral. In *Proc. NSDI*, 2004.
- [10] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, November 1996. http://wp.netscape.com/eng/ssl3/.
- [11] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 2983, 2000.
- [12] S. Gupta and A. L. M. Reddy. A Client Oriented, IP Level Redirection Mechansism. In *Proc. IEEE INFO-COM*, 1999.
- [13] T. Henderson. Generalizing the HIP Base
 Protocol, 2005. http://www.join.
 uni-muenster.de/Dokumente/drafts/
 draft-henderson-hip-generalize-00.txt.
- [14] T. Henderson and A. Gurtov. HIP Experiment Report, 2005. http://
 www.ietf.org/internet-drafts/
 draft-irtf-hip-experiment-01.txt.
- [15] Internet protocol v4 adress space. http://www.iana.org/assignments/.
- [16] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *Proc. of IPTPS*, 2004.
- [17] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, 2000.
- [18] H. V. Madhyastha, A. Venkataramani, A. Krishnamurthy, and T. Anderson. Oasis: An Overlay-aware Network Stack. SIGOPS Operating Systems Review, 40(1), 2006.
- [19] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol, 2003. http: //www.hip4inter.net/documentation/ drafts/draft-moskowitz-hip-08.html.
- [20] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. *IEEE Communications Magazine*, 2001.
- [21] A. Nakao, L. Peterson, and M. Wawrzoniak. A Divert Mechanism for Service Overlays. Technical Report TR-668-03, CS Dept, Princeton, Feb 2003.
- [22] T. S. E. Ng, I. Stoica, and H. Zhang. A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces. In *Proc. USENIX*, 2001.

- [23] Global Environment for Networking Investigations (GENI). http://www.nsf.gov/cise/geni.
- [24] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.
- [25] C. Perkins. IP Mobility Support. RFC 2002, 2002.
- [26] Planet Lab. http://www.planet-lab.org.
- [27] J. Postel. Echo Protocol. RFC 862, 1983.
- [28] P. Rodriguez, S. Mukherjee, and S. Rangarajan. Session-level Techniques for Improving Web Browsing Performance on Wireless Links. In Proc. of the 13th international conference on World Wide Web, 2004.
- [29] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A Case for Informed Internet Routing and Transport. Technical Report TR-98-10-05, 1998.
- [30] A. Seth, P. Darragh, and S. Keshav. A Generalized Architecture for Tetherless Computing in Disconnected Networks. http://mindstream.watsmore.net/.
- [31] E. Shi, D. Andersen, A. Perrig, and I. Stoica. Over-DoSe: A Generic DDoS Solution Using an Overlay Network. Technical Report CMU-CS-06-114, Carnegie Mellon University, 2006.
- [32] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In SIG-COMM, 2002.
- [33] G. Su. MOVE: Mobility with Persistent Network Connections. PhD thesis, Columbia University, Oct 2004.
- [34] G. Su and J. Nieh. Mobile Communication with Virtual Network Address Translation. Technical Report CUCS-003-02, Columbia University, Feb 2002.
- [35] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay-based Architecture for Enhancing Internet QoS. In *Proc. of NSDI*, 2004.
- [36] F. Teraoka, Y. Yokote, and M. Tokoro. A Network Architecture Providing Host Migration Transparency. In *Proc. ACM SIGCOMM*, 1991.
- [37] Virtual private network consortium. http://www. vpnc.org/.
- [38] Virtual tunnel. http://vtun.sourceforge. net/.
- [39] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *Proc. of OSDI*, 2004.
- [40] K. Wehrle, F. Pahlke, D. Muller, et al. Linux Networking Architecture: Design and Implementation of Networking Protocols in the Linux Kernel, 2004. Prentice-Hall.
- [41] J. Wrocławski. The MetaNet: White Paper. In Workshop on Research Directions for the Next Generation Internet, 1997.
- [42] P. Yalagandula, A. Garg, M. Dahlin, L. Alvisi, and H. Vin. Transparent Mobility with Minimal Infrastructure. Technical Report TR-01-30, UT Austin, June 2001.
- [43] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proc. of MOBISYS*, 2003.

Distributed Quota Enforcement for Spam Control

Michael Walfish, J.D. Zamfirescu*, Hari Balakrishnan*, David Karger*, and Scott Shenker

Abstract

Spam, by overwhelming inboxes, has made email a less reliable medium than it was just a few years ago. Spam filters are undeniably useful but unfortunately can flag non-spam as spam. To restore email's reliability, a recent spam control approach grants quotas of stamps to senders and has the receiver communicate with a wellknown quota enforcer to verify that the stamp on the email is fresh and to cancel the stamp to prevent reuse. The literature has several proposals based on this general idea but no complete system design and implementation that: scales to today's email load (which requires the enforcer to be distributed over many hosts and to tolerate faults in them), imposes minimal trust assumptions, resists attack, and upholds today's email privacy. This paper describes the design, implementation, analysis, and experimental evaluation of DQE, a spam control system that meets these challenges. DQE's enforcer occupies a point in the design spectrum notable for simplicity: mutually untrusting nodes implement a storage abstraction but avoid neighbor maintenance, replica maintenance, and heavyweight cryptography.

1 Introduction

Email is a less reliable communication medium than it was just a few years ago. The culprit is spam (defined as unsolicited bulk email), which drowned inboxes, making it hard for users to see the email they cared about. Unfortunately, spam filters, which offer inboxes muchneeded relief, have not restored reliability to email: false positives from filters are now a dominant mode of email failure. Anecdotal evidence suggests that the rate of false positives is 1% [11, 46], with some estimating their economic damage at hundreds of millions of dollars annually [12, 19]. While we have no way to verify these numbers, we can vouch for the personal inconvenience caused by false positives. And while our purpose here is not to cast aspersions on spam filters (indeed, we personally rely on them), we have nonetheless been vexed by what seems to be the inherent unreliability of contentbased spam control.

Instead, we turn to an approach using *quotas* or *bankable postage*, where the goal is to limit the number of messages sent, not divine their intent. Several such schemes have been proposed before [1, 5, 36]. In general, these systems give every sender a *quota* of *stamps*.

How this quota is determined varies among proposals; options include proof of CPU or memory cycles [1, 47], annual payment [5], having an email account with an ISP [36], having a driver's license, etc. The sending host or its email server attaches a stamp to each email message, and the receiving host or its email server tests the incoming stamp by asking a quota enforcer whether the enforcer has seen the stamp before. If not, the receiving host infers that the stamp is "fresh" and then cancels it by asking the enforcer to store a record of the stamp. The receiving host delivers only messages with fresh stamps to the human user; messages with used stamps are assumed to be spam. The hope is that allocating reasonable quotas to everyone and then enforcing those quotas would cripple spammers, who need huge volumes to be profitable, while leaving legitimate users largely unaffected; see §8.2 for a basic economic analysis.1

Of course, many defenses against spam have been proposed, each with advantages and disadvantages. The purpose of this paper is not to claim that ours is superior to all others or that its adoption will be easy. Rather, the purpose is to prove that many technical hurdles in quota-based systems, described below, can be overcome. To that end, this paper describes the design, implementation, analysis, and experimental evaluation of *DQE* (Distributed Quota Enforcement), a quota-based spam control system.

To be viable, DQE must meet two sets of design goals (see §2). The first set concerns the protocol between receivers and the enforcer. The protocol must never flag messages with fresh stamps as spam, must preserve the privacy of sender-receiver communication, and must not require that email servers and clients trust the enforcer. The second set applies to the enforcer: it must scale to current and future email volumes, requiring distribution over many machines, perhaps across several organizations; it must allow faults without letting much spam through; and it must resist attacks. Also, the enforcer should tolerate mutual mistrust among its constituent hosts (which is separate from the requirement, stated above, that the enforcer not be trusted by its clients). Finally, the enforcer should achieve high throughput to minimize management and hardware costs. Previous proposals do not meet these requirements (see §7.1).

Our main focus in this paper is the quota enforcer, which serves as a "clearing house" for canceled stamps. The enforcer stores billions of key-value pairs (canceled

^{*}MIT Computer Science and AI Lab

[†]UC Berkeley and ICSI

stamps) over a set of mutually untrusting nodes and tolerates Byzantine and crash faults. It relies on just one trust assumption, common in distributed systems: that the constituent hosts are determined by a trusted entity (§4). The enforcer uses a replication protocol in which churn generates no extra work but which gives tight guarantees on the average number of reuses per stamp (§4.1). Each node uses an optimized internal key-value map that balances storage and speed (§4.2), and nodes shed load with a technique that avoids "distributed livelock" (§4.3).

Apart from these techniques, what is most interesting to us about the enforcer is its simplicity. By tailoring our solution to the semantics of quota enforcement (specifically, that the effect of lost data is only that spammers' effective quotas increase), we can meet the various design challenges with an infrastructure in which the nodes need neither keep track of other nodes, nor perform replica maintenance, nor use heavyweight cryptography.

In part because of this simplicity, the enforcer is practical. We have a deployed (though lightly used) system, and our experimental results suggest that our implementation can handle the world's email volume—over 80 billion messages daily [30, 50]—with a few thousand dedicated high-end PCs (§6).

This work is preceded by a workshop paper [5]. That paper argued, and this paper concurs, that quota *allocation* and *enforcement* should be separate. That paper proposed a receiver-enforcer protocol that DQE incorporates, but it sketched a very different (and more complex) enforcer design based on distributed hash tables (DHTs).

2 Requirements and Challenges

In this section we discuss general requirements for DQE and specific challenges for the enforcer. These requirements all concern quota *enforcement*; indeed, in this paper we address quota *allocation* only briefly (see §8). The reason for this focus is that these two are different concerns: the former is a purely technical matter while the latter involves social, economic, and policy factors.

2.1 Protocol Requirements

No false positives Our high-level goal is reliable email. We assume reused stamps indicate spam. Thus, a fresh stamp must never appear to have been used before.

Untrusted enforcer We do not know the likely economic model of the enforcer, whether *monolithic* (*i.e.*, owned and operated by a single entity) or *federated* (*i.e.*, many organizations with an interest in spam control donate resources to a distributed system). No matter what model is adopted, it would be wise to design the system so that clients place minimal trust in the infrastructure.

Privacy To reduce (already daunting) deployment hurdles, we seek to preserve the current "semantics" of

email. In particular, queries of the quota enforcer should not identify email senders (otherwise, the enforcer knows which senders are communicating with which receivers, violating email's privacy model), and a receiver should not be able to use a stamp to prove to a third party that a sender communicated with it.

2.2 Challenges for the Enforcer

Scalability The enforcer must scale to current and future email volumes. Studies estimate that 80-90 billion emails will be sent daily this year [30, 50]. (We admit that we have no way to verify these claims.) We set an initial target of 100 billion daily messages (an average of about 1.2 million stamp checks per second) and strive to keep pace with future growth. To cope with these rates, the enforcer must be composed of many hosts.

Fault-tolerance Given the required number of hosts, it is highly likely that some subset will experience crash faults (*e.g.*, be down) or Byzantine faults (*e.g.*, become subverted). The enforcer should be robust to these faults. In particular, it should guarantee no more than a small amount of stamp reuse, despite such failures.

High throughput To control management and hardware costs, we wish to minimize the required number of machines, which requires maximizing throughput.

Attack-resilience Spammers will have a strong incentive to cripple the enforcer; it should thus resist denial-of-service (DoS) and resource exhaustion attacks.

Mutually untrusting nodes In both federated and monolithic enforcer organizations, nodes could be compromised. In the federated case, even when the nodes are uncompromised, they may not trust each other. Thus, in either case, besides being untrusted (by clients), nodes should also be untrusting (of other nodes), even as they do storage operations for each other.

We now show how the above requirements are met, first discussing the general architecture in §3 and then, in §4, focusing on the detailed design of the enforcer.

3 DQE Architecture

The architecture is depicted in Figure 1. This section describes the format and allocation of stamps (§3.1), how stamps are checked and canceled (§3.2), and how that process satisfies the requirements in §2.1.² We also give an overview of the enforcer (§3.3) and describe attackers and vulnerabilities (§3.4). Although we will refer to "sender" and "receiver", we expect those will be, for ease of deployment, the sender's and receiver's respective email servers.

3.1 Stamp Allocation and Creation

The quota allocation policy is the purview of a few globally trusted quota allocators (QAs), each with dis-

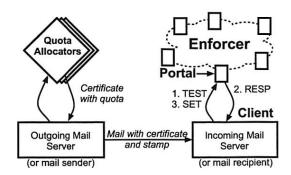


Fig. 1: DQE architecture.

tinct public/private key pair (QA_{pub}, QA_{priv}) ; the QA_{pub} are well known. A participant S constructs public/private key pair (S_{pub}, S_{priv}) and presents S_{pub} to a QA. The QA determines a quota for S and returns to S a signed certificate (the notation $\{A\}_B$ means that string A is signed with key B):

$$C_S = \{S_{pub}, expiration \ time, quota\}_{QA_{priv}}.$$

Anyone knowing QA_{pub} can verify, by inspecting C_S , that S has been allocated a quota. *expiration time* is when the certificate expires (in our implementation, certificates are valid for one year), and *quota* specifies the maximum number of stamps that S can use within a well-known epoch (in our implementation, each day is an epoch). Epochs free the enforcer from having to store canceled stamps for long time periods. Obtaining a certificate is the only interaction participants have with a QA, and it happens on, *e.g.*, yearly time scales, so the QA can allocate quotas with great care.

Participants use the *quota* attribute of their certificates to create up to *quota* stamps in any epoch. A participant with a certificate may give its stamps to other email senders, which may be a practical way for an organization to acquire a large quota and then dole it out to individual users.

Each stamp has the form $\{C_S, \{i, t\}_{S_{priv}}\}$. Each i in [1, quota] is supposed to be used no more than once in the current epoch. t is a unique identifier of the current epoch. Because email can be delayed en route to a recipient, receivers accept stamps from the current epoch and the one just previous.

An alternative to senders creating their own stamps would be QAs distributing stamps to senders. We reject this approach because it would require a massive computational effort by the QAs.

3.2 Stamp Cancellation Protocol

This section describes the protocol followed by senders, receivers, and the enforcer. Figure 2 depicts the protocol.

For a given stamp attached to an email from sender S, the receiver R must check that the stamp is unused and

- 1. S constructs STAMP = $\{C_S, \{i, t\}_{S_{priv}}\}$
- 2. $S \rightarrow R$: {STAMP, msg}.
- 3. R checks that $i \le quota$ (in C_S), that t is the current or previous epoch, that $\{i, t\}$ is signed with S_{priv} (S_{pub} is in C_S), and that C_S is signed with a quota allocator's key. If not, R rejects the message; the stamp is invalid. Otherwise, R computes POSTMARK = HASH(HASH(STAMP)).
- R → Enf.: TEST(POSTMARK). Enf. replies with x. If x is HASH(STAMP), R considers STAMP used. If x is "not found", R continues to step 5.
- 5. $R \rightarrow \text{Enf.}$: SET(POSTMARK, HASH(STAMP)).

Fig. 2: Stamp cancellation protocol followed by sender (S), receiver (R), and the enforcer (Enf.). The protocol upholds the design goals in §2.1: it gives no false positives, preserves privacy, and does not trust the enforcer.

must prevent reuse of the stamp in the current epoch. To this end, R checks that the value of i in the stamp is less than S's quota, that t identifies the current or just previous epoch, and that the signatures are valid. If the stamp passes these tests, R communicates with the enforcer using two UDP-based Remote Procedure Calls (RPCs): TEST and SET. R first calls TEST to check whether the enforcer has seen a fingerprint of the stamp; if the response is "not found", R then calls SET, presenting the fingerprint to be stored.³ The fingerprint of the stamp is HASH(STAMP), where HASH is a one-way hash function that is hard to invert.⁴

Note that an adversary cannot cancel a victim's stamp before the victim has actually created it: the stamp contains a signature, so guessing HASH(STAMP) requires either finding a collision in HASH or forging a signature.

We now return to the design goals in §2.1. First, false positives are impossible: because HASH is one-way, a reply of the fingerprint—HASH(STAMP)—in response to a TEST of the postmark—HASH(HASH(STAMP))—proves that the enforcer has seen the (postmark, fingerprint) pair. Thus, the enforcer cannot falsely cause an email with a novel stamp to be labeled spam. (The enforcer can, however, allow a reused stamp to be labeled novel; see §4.) Second, receivers do not trust the enforcer: they demand proof of reuse (*i.e.*, the fingerprint). Third, the protocol upholds current email privacy semantics: the enforcer sees hashes of stamps and not stamps themselves, so it doesn't know who sent the message. More details about this protocol's privacy properties are in [5].

3.3 The Enforcer

The enforcer stores the fingerprints of stamps canceled (*i.e.*, SET) in the current and previous epochs. It comprises thousands of untrusted *storage nodes* (which we

often call just "nodes"), with the list of approved nodes published by a trusted authority. The nodes might come either from a single organization that operates the enforcer for profit (perhaps paid by organizations with an interest in spam control) or else from multiple contributing organizations.

Clients, typically incoming email servers, interact with the enforcer by calling its interface, TEST and SET. These two RPCs are implemented by every storage node. For a given TEST or SET, the node receiving the client's request is called the *portal* for that request. Clients discover a nearby portal either via hard-coding or via DNS.

3.4 Attackers and Remaining Vulnerabilities

Attackers will likely be *spammers* (we include in this term both authors and distributors of spam). Attackers may control armies of hundreds of thousands of bots that can send spam and mount attacks.

As discussed in $\S 3.2$, attackers cannot forge stamps, cancel stamps they have not seen, or induce false positives. DQE's remaining vulnerabilities are in two categories: unauthorized stamp *use* (*i.e.*, theft) and stamp *re*use. We discuss the first category below. Since the purpose of the enforcer is to prevent reuse, we address the second one when describing the enforcer's design in $\S 4$.

A spammer may be able to steal stamps from its "botted" hosts. However, such theft by a single spammer is unlikely to increase spam much: a botnet with 100,000 hosts and a daily quota of 100 stamps per machine leads to 10 million extra spams, a small fraction of the tens of billions of daily spams today. Moreover, out-of-band contact between the email provider and the customer could thwart such theft, in analogy with credit card companies contacting customers to verify anomalous activity.

A related attack is to compromise an email relay and appropriate the fresh stamps on legitimate email. The attacker could then send more spam (but not much more—one relay is unlikely to carry much of the world's email). More seriously, the emails that were robbed now look like spam and might not be read. But, though the attack has greater appeal under DQE, the vulnerability is not new: even without stamps, an attacker controlling a compromised email relay can drop email arriving at the relay. In any case, encrypting emails could prevent stamp theft.

4 Detailed Design of the Enforcer

The enforcer, depicted in Figure 3, is a high-throughput storage service that replicates immutable key-value pairs over a group of mutually untrusting, infrequently changing nodes. It tolerates Byzantine faults in these nodes. We assume a trusted *bunker*, an entity that communicates the system membership to the enforcer nodes. The bunker assigns random *identifiers*—whose purpose we describe below—to each node and infrequently (*e.g.*, daily) dis-

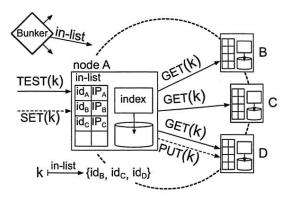


Fig. 3: Enforcer design. A TEST causes multiple GETs; a SET causes one PUT. Here, A is the portal. The ids are in a circular identifier space with the identifiers determined by the bunker.

tributes to each node an *in-list*, a digitally signed, authoritative list of the members' identifiers and IP addresses.

Given the required size of the system—thousands of nodes (§6.5)—we believe the bunker is a reasonable assumption. If a single organization operates the enforcer, the bunker can be simply the human who deploys the machines. If the enforcer is federated, a small number of neutral people can implement the bunker. Managing a list of several thousand relatively reliable machines that are donated by various organizations is a "human scale" job, and the vetting of machines can be light since the enforcer is robust to adversarial nodes. Of course, the bunker is a single point of vulnerability, but observe that humans, not computers, execute most of its functions. Nevertheless, to guard against a compromised bunker, nodes accept only limited daily changes to the in-list.

Clients' queries—e.g., TEST(HASH(HASH(stamp)))— are interpreted by the enforcer as queries on key-value pairs, *i.e.*, as TEST(k) or SET(k, ν), where k = HASH(ν). (Throughout, we use k and ν to mean keys and values.)

Portals implement TEST and SET by invoking at other nodes a UDP-based RPC interface, internal to the enforcer, of GET(k) and PUT(k, v). (Although the enforcer uses consistent hashing [33] to assign key-value pairs to nodes, which is reminiscent of DHTs, the enforcer and DHTs have different structures and different goals; see §7.2.) To ensure that GET and PUT are invoked only by other nodes, the in-list can include nodes' public keys, which nodes can use to establish pairwise shared secrets for lightweight packet authentication (e.g., HMAC [35]).

The rest of this section describes the detailed design of the enforcer. We first specify TEST and SET and show that even with crash failures (*i.e.*, down or unreachable nodes), the enforcer guarantees little stamp reuse. We then show how nodes achieve high throughput with an efficient implementation of PUT and GET ($\S4.2$) and a way to avoid degrading under load ($\S4.3$). We then consider attacks *on* nodes ($\S4.4$) and attacks *by* nodes, and we argue that a Byzantine failure reduces to a crash fail-

```
procedure TEST(k)
 v \leftarrow GET(k) // local check
 if v \neq "not found" then return (v)
 // r assigned nodes determined by in-list
 nodes \leftarrow ASSIGNED\_NODES(k)
 for each n \in nodes do {
      v \leftarrow n.GET(k) // invoke RPC
     // if RPC times out, continue
      if v \neq "not found" then return (v)
 // all nodes returned "not found" or timed out
 return ("not found")
procedure SET(k, v)
 PUT(k, v) // local store
 nodes \leftarrow ASSIGNED\_NODES(k)
 n \leftarrow \text{choose random } n \in nodes
 n.PUT(k, v) // invoke RPC
```

Fig. 4: Pseudo-code for TEST and SET in terms of GET and PUT.

ure in our context (§4.5). Our design decisions are driven by the challenges in §2.2, but the map between them is not clean: multiple challenges are relevant to each design decision, and vice versa.

4.1 TEST, SET, and Fault-Tolerance

Each key k presented to a portal in TEST or SET has r assigned nodes that could store it; these nodes are a "random" subset (determined by k) of enforcer nodes. We say below how to determine r. To implement TEST(k), a portal invokes GET(k) at k's r assigned nodes in turn. The portal stops when either a node replies with a v such that k = HASH(v), in which case the portal returns v to its client, or else when it has tried all r nodes without such a reply, in which case the portal returns "not found". To implement SET(k, v), the portal chooses one of the r assigned nodes uniformly at random and invokes PUT(k, v) there. Pseudo-code for TEST and SET is shown in Figure 4. The purpose of 1 PUT and r GETs—as opposed to the usual r PUTs and 1 GET—is to conserve storage.

A key's assigned nodes are determined by consistent hashing [33] in a circular identifier space using r hash functions. The bunker-given identifier mentioned above is a random choice from this space. To achieve near-uniform per-node storage with high probability, each node actually has multiple identifiers [61] deterministically derived from its bunker-given one.

Churn Churn generates no extra work for the system. To handle *intra-day* churn (*i.e.*, nodes going down and coming up between daily distributions of the in-list), portals do not track which nodes are up; instead they apply to each PUT or GET request a timeout of several seconds with no retry, and interpret a timed-out GET as simply a "not found". (A few seconds of latency is not problem-

atic for the portal's client—an incoming email server—because sender-receiver latency in email is often seconds and sometimes minutes.) Moreover, when a node fails, other nodes do not "take over" the failed node's data: the invariant "every (k, v) pair must always exist at r locations" is not needed for our application.

To handle *inter-day* churn (*i.e.*, in-list changes), the assigned nodes for most (k, v) pairs must not change; otherwise, queries on previously SET stamps (*e.g.*, "yesterday's" stamps) would fail. This requirement is satisfied because the bunker makes only minor in-list changes from day-to-day and because, from consistent hashing, these minor membership changes lead to proportionately minor changes in the assigned nodes [33].

Analysis We now show how to set r to prevent significant stamp reuse. We will assume that nodes, even subverted ones, do not abuse their *portal* role; we revisit this assumption in §4.5.

Our analysis depends on a parameter p, the fraction of the n total machines that fail during a 2-day period (recall that an epoch is a day and that nodes store stamps' fingerprints for the current and previous epochs). We will consider only a stamp's expected reuse. A Chernoff bound (proof elided) can show that there is unlikely to be a set of pn nodes whose failure would result in much more than the expected stamp reuse.

We don't distinguish the causes of failures—some machines may be subverted, while others may simply crash. To keep the analysis simple, we also do not characterize machines as reliable for some fraction of the time—we simply count in p any machine that fails to operate perfectly over the 2-day period. Nodes that do operate perfectly (i.e., remain up and follow the protocol) during this period are called good. We believe that carefully chosen nodes can usually be good so that p = 0.1, for example, might be a reasonably conservative estimate. Nevertheless, observe that this model is very pessimistic: a node that is offline for a few minutes is no longer good, yet such an outage would scarcely increase total spam.

For a given stamp, portals can detect attempted reuses once the stamp's fingerprint is PUT on a good node. When most nodes are good, this event happens quickly. As shown in the appendix, the expected number of times a stamp is used before this event happens is less than $\frac{1}{1-2p}+p^rn$. The second term reflects the possibility (probability p^r) that none of the r assigned nodes is good. In this case, an adversary can reuse the stamp once for each of the n portals. (The "local PUT" in the first line of SET in Figure 4 prevents infinite reuse.) These "lucky" stamps do not worry us: our goal is to keep small the total number of reuses across all stamps. If we set $r=1+\log_{1/p}n$ and take p=0.1, then a stamp's expected number of uses is less than $\frac{1}{1-2p}+p\approx 1+3p=1.3$, close to the ideal of 1 use per stamp.

```
procedure GET(k)
 b \leftarrow \text{INDEX.LOOKUP}(k)
 if b == NULL then return ("not found")
 a \leftarrow \text{DISK.READ}(b) // array a gets disk block b
 if k \notin a then
                  // scan all keys in a
    return ("not found") // index gave false location
 else return (v) // v next to k in array a
procedure PUT(k, v)
 if HASH(v) \neq k then return ("invalid")
 b \leftarrow \text{INDEX.LOOKUP}(k)
 if b == NULL then
    b \leftarrow \text{DISK.WRITE}(k, v) // write is sequential
    // b is disk block where write happened
    INDEX.INSERT(k, b)
         // we think k is in block b
    a \leftarrow \text{DISK.READ}(b) // array a gets disk block b
                      // false location: k not in block b
    if k \notin a then
        b' \leftarrow \text{DISK.WRITE}(k, v)
        INDEX.OVERFLOW.INSERT(k, b')
```

Fig. 5: Pseudo-code for GET and PUT. A node switches between batches of writes and reads; that asynchrony is not shown.

The above assumes that the network never loses RPCs. To handle packet loss, clients and portals can retry RPCs, thereby lowering the effective drop rate and making the false negatives from dropped packets a negligible contribution to total spam. Investigating whether such retries are necessary is future work.

4.2 Implementation of GET and PUT

In our early implementation, nodes stored their internal key-value maps in memory, which let them give fast "found" and "not found" answers to GETs. However, we realized that the total number of stamps that the enforcer must store makes RAM scarce. Thus, nodes need a way to store keys and values that conserves RAM yet, as much as possible, allows high PUT and GET throughput.

This section describes the nodes' key-value stores, the properties of which are: PUTs are fast; after a crash, nodes can recover most previously canceled stamps; each key-value pair costs 5.2 bytes rather than 40 bytes of RAM; "not found" answers to GETs are almost always fast; and "found" answers to GETs require a disk seek. We justify these properties below.

As in previous systems [39,49,54], nodes write incoming data—key-value pairs here—to a disk log sequentially and keep an index that maps keys to locations in the log. In our system, the index lives in memory and maps keys to log *blocks*, each of which contains multiple key-value pairs. Also, our index can return *false locations*: it occasionally "claims" that a given key is on the disk even though the node has never stored the key.

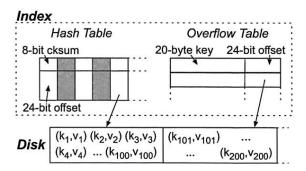


Fig. 6: In-RAM index mapping from k to log block that holds (k, v).

When a node looks up a key k, the index returns either "not stored" or a block b. In the latter case, the node reads b from the on-disk log and scans the keys in b to see if k is indeed stored. Pseudo-code describing how GETs and PUTs interact with the index is shown in Figure 5.

We now describe the structure of the index, depicted in Figure 6. The index has two components. First is a modified open addressing hash table, the entries of which are divided into an 8-bit checksum and a 24-bit pointer to a block (of size, e.g., 4 KBytes). A key k, like in standard open addressing as described by Knuth, "determines a 'probe sequence,' namely a sequence of table positions that are to be inspected whenever k is inserted or looked up" [34], with insertion happening in the first empty position. When insertion happens, the node stores an 8-bit checksum of k as well as a pointer to the block that holds k. (The checksum and probe sequence should be unpredictable to an adversary.) A false location happens when a lookup on key k finds an entry for which the top 8 bits are k's checksum while the bottom bits point to a block that does not hold k. This case is handled by the index's second component, an *overflow table* storing those (k, v)pairs for which k wrongly appears to be in the hash table. INDEX.LOOKUP(), in Figure 5, checks this table.

We now return to the properties claimed above. PUTs are fast because the node, rather than interleaving reads and writes, does each in batches, yielding sequential disk writes. For crash recovery: on booting, a node scans its log to rebuild the index. For the RAM cost: the value of the hash table's load factor (i.e., ratio of non-empty entries to total entries) that is space-minimizing is ≈ 0.87 (see Claim 1 in [66]); the corresponding RAM cost is 1.3x entries (see Claim 2 in [66]), where x is the number of (k, v) pairs stored by the node. The 1.3x entries with 4 bytes per entry gives the 5.2 bytes claimed above. For negative GET(k) requests (i.e., k not found), nodes inspect an average of 8 entries in the probe sequence (see Claim 3 in [66]), and the rare false location incurs a disk seek. For affirmative GETs (i.e., reused stamps), the node visits an average of 8 entries to look up the block, b, that holds v; the node then does a disk seek to get b.

These seeks are one of the enforcer's principal bottlenecks, as shown in $\S 6.3$. To ease this bottleneck, nodes cache recently retrieved (k, ν) pairs in RAM.

Nodes use the block device interface rather than the file system. With the file system, the kernel would, on retrieving a (k, v) pair from disk, put in its buffer cache the entire disk block holding (k, v). However, most of that cached block would be a waste of space: nodes' disk reads exhibit no reference locality.

4.3 Avoiding "Distributed Livelock"

The enforcer must not degrade under high load. Such load could be from heavy legitimate use or from attackers' spurious requests, as in §4.4. In fact, our implementation's capacity, measured by total correct TEST responses, did originally worsen under load. This section describes our change to avoid this behavior. See §6.6 for experimental evidence of the technique's effectiveness.

Observe that the packets causing nodes to do work are UDP RPC requests or responses and that these packets separate into three classes. The classes are: (1) TEST or SET requests from clients; (2) GET or PUT requests from other enforcer nodes; and (3) GET or PUT responses. To achieve the enforcer's throughput goal, which is to maximize the number of successful PUTs and GETs, we have the individual nodes *prioritize these packet classes*. The highest priority class is (3), the lowest (1).

When nodes did not prioritize and instead served these classes round-robin, *overload*—defined as the CPU being unable to do the work induced by all arriving packets—caused two problems. First, each packet class experienced drops, so many GETs and PUTs were unsuccessful since either the request or the response was dropped. Second, the system admitted too many TESTs and SETs, *i.e.*, it overcommitted to clients. The combination was *distributed* livelock: nodes spent cycles on TESTs and SETs and meanwhile dropped GET and PUT requests and responses from *other* nodes.

Prioritizing the three classes, in contrast to roundrobin, improves throughput and implements admission control: a node, in its role as portal, commits to handling a TEST or SET only if it has no other pending work in its role as node. We can view the work induced by a TEST or SET as a *distributed* pipeline; each stage is the arrival at *any* node of a packet related to the request. In this view, a GET or PUT response means the enforcer as a *whole* has done most of the work for the underlying request; dropping such a packet contradicts the throughput goal.

To implement the priorities, each of the three packet classes goes to its own UDP destination port and thus its own queue (socket) on the node. The node reads from the highest priority queue (socket) with data. If the node cannot keep up with a packet class, the associated socket buffer fills, and the kernel drops packets in that class.

A different way to avoid distributed livelock might be for a node to maintain a window of outstanding RPCs to every other node. This approach will not work well in general because it is hard to set the size of the window. We also note that avoiding distributed livelock and coping with network congestion are separate concerns; we briefly address the latter in §4.6.

The general approach described in this section—which does nothing more than apply the principle that, under load, one should drop from the beginning of a pipeline to maximize throughput—could be useful for other distributed systems. There is certainly much work addressing overload: see, e.g., SEDA [68, 69], LRP [15], and Defensive Programming [48] and their bibliographies; these proposals use fine-grained resource allocation to protect servers from overload. Other work (see, e.g., Neptune [57] and its bibliography) focuses on clusters of equivalent servers, with the goal of proper allocation of requests to servers. All of this research concerns requests of single hosts and is orthogonal to the simple priority scheme described here, which concerns logical requests happening on several hosts.

4.4 Resource Exhaustion Attacks

Two years ago, a popular DNS-based block list (DNSBL) was forced offline [27], and a few months later another such service was attacked [63], suggesting that effective anti-spam services with open interfaces are targets for various denial-of-service (DoS) attacks. If successful, DQE would be a major threat to spammers, so we must ensure that the enforcer resists attack. We do not focus on *packet floods*, in which zombies [51,56] exhaust the enforcer's bandwidth with packets that are not well-formed requests. These attacks can be handled using various commercial (*e.g.*, upstream firewalls) and academic (see [44] for a survey) solutions. We thus assume that enforcer nodes see only well-formed RPC requests.

A resource exhaustion attack is a flood of spurious RPCs (e.g., by zombies). Such floods would waste nodes' resources, specifically: disk seeks on affirmative GETs, entries in the RAM index (which is exhausted long before the disk fills) for PUTs, and CPU cycles to process RPCs. These attacks are difficult because one cannot differentiate "good" from "bad": requests are TEST(k) and SET(HASH(v), v) where k, v are any 20-byte values. Absent further mechanism, handling such an attack requires the enforcer to be provisioned for the legitimate load plus as many TESTs and SETs as the attacker can send.

Before we describe the defense, observe that attackers have *some* bandwidth limit. Let us make the assumption—which we revisit shortly—that attackers are *sending as much spam as they can*, and, specifically, that they are limited by bandwidth. This limit reflects either a constraint like the bots' access links or some

threshold above which the attacker fears detection by the human owner of the compromised machine.

Observe, also, that the enforcer is indifferent between the attacker sending (1) a spurious TEST and (2) a single spam message, thereby inducing a legitimate TEST (and, rarely, a SET); the resources consumed by the enforcer are the same in (1) and (2). Now, under the assumption above, we can neutralize resource exhaustion attacks by arranging for a TEST or SET to require the same amount of bandwidth as sending a spam. For if attackers are "maxed out" and if sending a TEST and a spam cost the same bandwidth, then attackers cannot cause more TESTs and SETs than would be induced anyway by current email volumes—for which the enforcer is already provisioned. To realize this general approach (which is in the spirit of [58, 65]), enforcer nodes have several options, such as asking for long requests or demanding many copies of each request. This approach does not address hotspots (i.e., individual, overloaded portals), but if any particular portal is attacked, clients can use another one.

Of course, despite our assumption above, today's attackers are unlikely to be "maxed out". However, they have *some* bandwidth limit. If this limit and current spam volumes are the same order of magnitude, then the approach described here reduces the enforcer's required over-provisioning to a small constant factor. Moreover, this over-provisioning is an upper bound: the most damaging spurious request is a TEST that causes a disk seek by asking a node for an existing stamp fingerprint (§6.3), yet nodes cache key-value pairs (§4.2). If, for example, half of spurious TESTs generate cache hits, the required provisioning halves.

4.5 Adversarial Nodes

We now argue that for the protocol described in §4.1, a Byzantine failure reduces to a crash failure. Nodes do not route requests for each other. A node cannot lie in response to GET(k) because for a false ν , $HASH(\nu)$ would not be k (so a node cannot make a fresh stamp look reused). A node's only attack is to cause a stamp to be reused by ignoring PUT and GET requests, but doing so is indistinguishable from a crash failure. Thus, the analysis in §4.1, which applies to crash failures, captures the effect of adversarial nodes. Of course, depending on the deployment (federated or monolithic), one might have to assume a higher or lower p.

However, the analysis does not cover a node that abuses its portal role and endlessly gives its clients false negative answers, letting much spam through. Note, though, that if adversarial portals are rare, then a random choice is unlikely to find an adversarial one. Furthermore, if a client receives much spam with apparently fresh stamps, it may become suspicious and switch portals, or it can query multiple portals.

Another attack for an adversarial node is to execute spurious PUTs and GETs at other nodes, exhausting their resources. In defense, nodes maintain "put quotas" and "get quotas" for each other, which relies on the fact that the assignment of (k, v) pairs to nodes is balanced. Deciding how to set these quotas is future work.

4.6 Limitations

The enforcer may be either clustered or wide-area. Because our present concern is throughput, our implementation and evaluation are geared only to the clustered case. We plan to address the wide-area case in future work and briefly consider it now. If the nodes are separated by low capacity links, distributed livelock avoidance (§4.3) is not needed, but congestion control is. Options include long-lived pairwise DCCP connections or a scheme like STP in Dhash++ [14].

5 Implementation

We describe our implementation of the enforcer nodes and DQE client software; the latter runs at email senders and receivers and has been handling the inbound and outbound email of several users for over six months.

5.1 Enforcer Node Software

The enforcer is a 5000-line event-driven C++ program that exposes its interfaces via XDR RPC over UDP. It uses libasync [42] and its asynchronous I/O daemon [39]. We modified libasync slightly to implement distributed livelock avoidance (§4.3). We have successfully tested the enforcer on Linux 2.6 and FreeBSD 5.3. We play the bunker role ourselves by configuring the enforcer nodes with an in-list that specifies random identifiers. We have not yet implemented per-portal quotas to defend against resource exhaustion by adversarial nodes (§4.5), a defense against resource exhaustion by clients (§4.4), or HMAC for inter-portal authentication (§4). The implementation is otherwise complete.

5.2 DQE Client Software

The DQE client software is two Python modules. The sender module is invoked by a sendmail hook; it creates a stamp (using a certificate signed by a virtual quota allocator) and inserts it in a new header in the departing message. The receiver module is invoked by procmail; it checks whether the email has a stamp and, if so, executes a TEST RPC over XDR to a portal. Depending on the results (no stamp, already canceled stamp, forged stamp, etc.), the module adds a header to the email for processing by filter rules. To reduce client-perceived latency, the module first delivers email to the recipient and then, for fresh stamps, asynchronously executes the SET.

6 Evaluation of the Enforcer

In this section, we evaluate the enforcer experimentally. We first investigate how its observed fault-tolerance—

The analysis (§4.1, appendix) accurately reflects how actual failures affect observed stamp reuse. Even with 20% of the nodes down, the average number of reuses is under 1.5.	§6.2
Microbenchmarks (§6.3) predict the enforcer's performance exactly. The bottleneck is disk seeks.	§6.4
The enforcer can handle current email volume with a few thousand high-end PCs.	§6.5
The scheme to avoid livelock (§4.3) is effective.	§6.6

Table 1: Summary of evaluation results.

in terms of the average number of stamp reuses as a function of the number of faulty machines—matches the analysis in §4.1. We next investigate the capacity of a single enforcer node, measure how this capacity scales with multiple nodes, and then estimate the number of dedicated enforcer nodes needed to handle 100 billion emails per day (our target volume; see §2.2). Finally, we evaluate the livelock avoidance scheme from §4.3. Table 1 summarizes our results.

All of our experiments use the Emulab testbed [18]. In these experiments, between one and 64 enforcer nodes are connected to a single LAN, modeling a clustered network service with a high-speed access link.

6.1 Environment

Each enforcer node runs on a separate Emulab host. To simulate clients and to test the enforcer under load, we run up to 25 instances of an open-loop tester, *U* (again, one per Emulab host). All hosts run Linux FC4 (2.6 kernel) and are Emulab's "PC 3000s", which have 3 GHz Xeon processors, 2 GBytes of RAM, 100 Mbit/s Ethernet interfaces, and 10,000 RPM SCSI disks.

Each U follows a Poisson process to generate TESTs and selects the portal for each TEST uniformly at random. This process models various email servers sending TESTs to various enforcer nodes. (As argued in [45], Poisson processes appropriately model a collection of many random, unrelated session arrivals in the Internet.) The proportion of reused TESTs (stamps 5 previously SET by U) to fresh TESTs (stamps never SET by U) is configurable. These two TEST types model an email server receiving a spam or non-spam message, respectively. In response to a "not found" reply—which happens either if the stamp is fresh or if the enforcer lost the reused stamp—U issues a SET to the portal it chose for the TEST.

Our reported experiments run for 12 or 30 minutes. Separately, we ran a 12-hour test to verify that the performance of the enforcer does not degrade over time.

6.2 Fault Tolerance

We investigate whether failures in the implemented system reflect the analysis. Recall that this analysis (in $\S4.1$ and the appendix) upper bounds the average number of stamp uses in terms of p, where p is the probability a

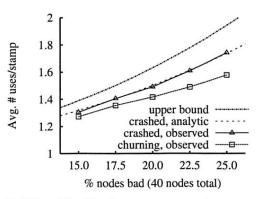


Fig. 7: Effect of "bad" nodes on stamp reuse for two types of "bad". Observed uses obey the upper bound from the analysis (see §4.1 and the appendix). The crashed case can be analyzed exactly; the observations track this analysis closely.

node is bad, i.e., that it is ever down while a given stamp is relevant (two days). Below, we model "bad" with crash faults, only (see §4.5 for the relationship between Byzantine and crash faults).

We run two experiments in which we vary the number of bad nodes. These experiments measure how often the enforcer—because some of its nodes have crashed—fails to "find" stamps it has already "heard" about.

In the first experiment, called *crashed*, the bad nodes are never up. In the second, called *churning*, the bad nodes repeat a 90-second cycle of 45 seconds of down time followed by 45 seconds of up time. Both experiments run for 30 minutes. The Us issue TESTs and SETs to the up nodes, as described in §6.1. Half the TESTs are for fresh stamps, and the other half are for a *reuse group*—843,750 reused stamps that are each queried 32 times during the experiment. This group of TESTs models an adversary trying to reuse a stamp. The Us count the number of "not found" replies for each stamp in the reuse group; each such reply counts as a stamp use. We set n = 40, and the number of bad nodes is between 6 and 10, so p varies between 0.15 and 0.25. For the replication factor (§4.1), we set r = 3.

The results are depicted in Figure 7. The two "observed" lines plot the average number of times a stamp in the "reuse group" was used successfully. These observations obey the model's least upper bound. This bound, from equation (1) in the appendix, is $1 + \frac{3}{2}p + 3p^2 + p^3 \left[40(1-p) - \left(1 + \frac{3}{2} + 3\right)\right]$ and is labeled "upper bound". The *crashed* experiment is amenable to an exact expectation calculation. The resulting expression sidepicted by the line labeled "crashed, analytic"; it matches the observations well.

6.3 Single-node Microbenchmarks

We now examine the performance of a single-node enforcer. We begin with RAM and ask how it limits the

Operation	Ops/sec	bottleneck
PUT	1,100	RAM
pessimistic GET	400	disk
non-pessimistic GET	38,000	CPU

Table 2: Single-node performance, assuming 1 GByte of RAM.

number of PUTs. Each key-value pair consumes roughly 5.2 bytes of memory in expectation (§4.2), and each is stored for two days (§3.3). Thus, with one GByte of RAM, a node can store slightly fewer than 200 million key-value pairs, which, over two days, is roughly 1100 PUTs per second. A node can certainly accept a higher average rate over any given period but must limit the total number of PUTs it accepts each day to 100 million for every GByte of RAM. Our implementation does not currently rate-limit inbound PUTs.

We next ask how the disk limits GETs. (The disk does not bottleneck PUTs because writes are sequential and because disk space is ample.) Consider a key k requested at a node d. We call a GET slow if d stores k on disk (if so, d has an entry for k in its index) and k is not in d's RAM cache (see §4.2). We expect d's ability to respond to slow GETs to be limited by disk seeks. To verify this belief, an instance of U sends TESTs and SETs at a high rate to a single-node enforcer, inducing local GETs and PUTs. The node runs with its cache of key-value pairs disabled. The node responds to an average of 400 slow GETs per second (measured over 5-second intervals, with standard deviation less than 10% of the mean). This performance agrees with our disk benchmark utility, which does random access reads in a tight loop.

We next consider fast GETs, which are GETs on keys k for which the node has k cached or is not storing k. In either case, the node can reply quickly. For this type of GET, we expect the bottleneck to be the CPU. To test this hypothesis, U again sends many TESTs and SETs. Indeed, CPU usage reaches 100% (again, measured over 5-second intervals with standard deviation as above), after which the node can handle no more than 38,000 RPCs. A profile of our implementation indicates that the specific CPU bottleneck is malloc().

Table 2 summarizes the above findings.

6.4 Capacity of the Enforcer

We now measure the capacity of multiple-node enforcers and seek to explain the results using the microbench-marks just given. We define capacity as the maximum rate at which the system can respond correctly to the reused requests. Knowing the capacity as a function of the number of nodes will help us, in the next section, answer the dual question: how many nodes the enforcer must comprise to handle a given volume of email (assuming each email generates a TEST).

Of course, the measured capacity will depend on the workload: the ratio of fresh to reused TESTs determines whether RAM or disk is the bottleneck. The former TESTs consume RAM because the SETs that follow induce PUTs, while the latter TESTs may incur a disk seek.

Note that the resources consumed by a TEST are different in the multiple-node case. A TEST now generates r (or r-1, if the portal is an assigned node) GET RPCs, each of which consumes CPU cycles at the sender and receiver. A reused TEST still incurs only one disk seek in the entire enforcer (since the portal stops GETing once a node replies affirmatively).

32-node experiments We first determine the capacity of a 32-node enforcer. To emulate the per-node load of a several thousand-node deployment, we set r = 5 (which we get because, from §4.1, $r = 1 + \log_{1/p} n$; we take p = 0.1 and n = 8000, which is the upper bound in §6.5).

We run two groups of experiments in which 20 instances of U send half fresh and half reused TESTs at various rates to this enforcer. In the first group, called disk, the nodes' LRU caches are disabled, forcing a disk seek for every affirmative GET ($\S4.2$). In the second group, called CPU, we enable the LRU caches and set them large enough that stamps will be stored in the cache for the duration of the experiment. The first group of experiments is fully pessimistic and models a disk-bound workload whereas the second is (unrealistically) optimistic and models a workload in which RPC processing is the bottleneck. We ignore the RAM bottleneck in these experiments but consider it at the end of the section.

Each node reports how many reused TESTs it served over the last 5 seconds (if too many arrive, the node's kernel silently drops). Each experiment run happens at a different TEST rate. For each run, we produce a value by averaging together all of the nodes' 5-second reports. Figure 8 graphs the positive response rate as a function of the TEST rate. The left and right y-axes show, respectively, a per-node per-second mean and a per-second mean over all nodes; the x-axis is the aggregate sent TEST rate. (The standard deviations are less than 9% of the means.) The graph shows that maximum per-node capacity is 400 reused TESTs/sec when the disk is the bottle-neck and 1875 reused TESTs/sec when RPC processing is the bottleneck; these correspond to 800 and 3750 total TESTs/sec (recall that half of the sent TESTs are reused).

The microbenchmarks explain these numbers. The per-node disk capacity is given by the disk benchmark. We now connect the per-node TEST-processing rate (3750 per second) to the RPC-processing microbenchmark (38,000 per second). Recall that a TEST generates multiple GET requests and multiple GET responses (how many depends on whether the TEST is fresh). Also, if the stamp was fresh, a TEST induces a SET request, a PUT request, and a PUT response. Taking all of these "re-

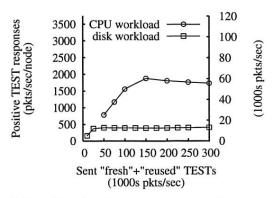


Fig. 8: For a 32-node enforcer, mean response rate to TEST requests as function of sent TEST rate for disk- and CPU-bound workloads. The two y-axes show the response rate in different units: (1) per-node and (2) over the enforcer in aggregate. Here, r = 5, and each reported sample's standard deviation is less than 9% of its mean.

quests" together (and counting responses as "requests" because each response also causes the node to do work), the average TEST generates 10.1 "requests" in this experiment (see [66] for details). Thus, 3750 TEST requests per node per second per second is 37,875 "requests" per node per second, which is within 0.5% of the microbenchmark from §6.3 (last row of Table 2).

One might notice that the CPU line in Figure 8 degrades after 1875 positive responses per second per node (the enforcer's RPC-processing capacity). The reason is as follows. Giving the enforcer more TESTs and SETs than it can handle causes it to drop some. Dropped SETs cause some future *reused* TESTs to be seen as *fresh* by the enforcer—but fresh TESTs induce more GETs (r or r-1) than reused TESTs (roughly (r+1)/2 on average since the portal stops querying when it gets a positive response). Thus, the degradation happens because extra RPCs from fresh-*looking* TESTs consume capacity. This degradation is not ideal, but it does not continue indefinitely.

Scaling We now measure the enforcer's capacity as a function of the number of nodes, hypothesizing nearlinear scaling. We run the same experiments as for 32 nodes but with enforcers of 8, 16, and 64 nodes. Figure 9 plots the maximum point from each experiment. (The standard deviations are smaller than 10% of the means.) The results confirm our hypothesis across this (limited) range of system sizes: an additional node at the margin lets the enforcer handle, depending on the workload, an additional 400 or 1875 TESTs/sec—the per-node averages for the 32-node experiment.

We now view the enforcer's scaling properties in terms of its request mix. Assume pessimistically that all reused TEST requests cost a disk seek. Then, doubling the rate of spam (reused TEST requests) will double the required enforcer size. However, doubling the rate of non-spam

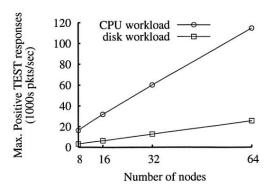


Fig. 9: Enforcer capacity under two workloads as a function of number of nodes in the enforcer. The y-axis is the same as the right-hand y-axis in Fig. 8. Standard deviations are smaller than 10% of the reported means.

100 billion	emails daily (target from §2.2)
65%	spam [7, 43]
65 billion	disk seeks / day (pessimistic)
400	disk seeks/second/node (§6.3)
86400	seconds/day
1881	nodes (from three quantities above)

Table 3: Estimate of enforcer size (based on average rates).

(fresh TEST requests) will not change the required enforcer size at first. The rate of non-spam will only affect the required enforcer size when the ratio of the rates of reused TESTs to fresh TESTs matches the ratio of a single node's performance limits, namely 400 reused TESTs/sec to 1100 fresh TESTs/sec for every GByte of RAM. The reason is that fresh TESTs are followed by SETs, and these SETs are a bottleneck only if nodes see more than 1100 PUTs per second per GByte of RAM; see Table 2.

6.5 Estimating the Enforcer Size

We now give a rough estimate of the number of dedicated enforcer nodes required to handle current email volumes. The calculation is summarized in Table 3. Some current estimates suggest 84 billion email messages per day [30] and a spam rate of roughly 65% [43]. (Brightmail reported a similar figure for the spam percentage in July 2004 [7].) We assume 100 billion messages daily and follow the lower bound on capacity in Figure 9, i.e., every reused TEST-each of which models a spam messagecauses the enforcer to do a disk seek. In this case, the enforcer must do 65 billion disk seeks per day and, since the required size scales with the number of disks (§6.4), a straightforward calculation gives the required number of machines. For the disks in our experiments, the number is about 2000 machines. The required network bandwidth is small, about 3 Mbits/s per node.

So far we have considered only average request rates. We must ask how many machines the enforcer needs to

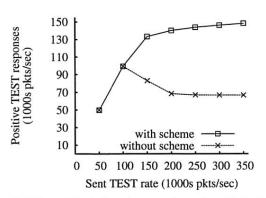


Fig. 10: Effect of livelock avoidance scheme from §4.3. As the sent TEST rate increases, the ability of an enforcer without the scheme to respond accurately to reused TESTs degrades.

handle peak email loads while bounding reply latency. To answer this question, we would need to determine the peak-to-average ratio of email reception rates at email servers (their workload induces the enforcer workload). As one data point, we analyzed the logs of our research group's email server, dividing a five-week period in early 2006 into 10-minute windows. The maximum window saw 4 times the volume of the average window. Separately, we verified with a 14-hour test that a 32-node enforcer can handle a workload of like burstiness with worst-case latency of 10 minutes. Thus, if global email is this bursty, the enforcer would need 8000 machines (the peak-to-average ratio times the 2000 machines derived above) to give the same worst-case latency.

However, global email traffic is likely far smoother than one server's workload. And spam traffic may be smoother still: the spam in [32]'s 2004 data exhibits—over ten minute windows, as above—a peak-to-average ratio of 1.9:1. Also, Gomes *et al.* [22] claim that spam is less variable than legitimate email. Thus, many fewer than 8000 machines may be required. On the other hand, the enforcer may need some over-provisioning for spurious TESTs (§4.4). For now, we conclude that the enforcer needs "a few thousand" machines and leave to future work a study of email burstiness and attacker ability.

6.6 Avoiding "Distributed Livelock"

We now briefly evaluate the scheme to avoid livelock (from $\S4.3$). The goal of the scheme is to maximize correct TEST responses under high load. To verify that the scheme meets this goal, we run the following experiment: 20 U instances send TEST requests (half fresh, half reused) at high rates, first, to a 32-node enforcer with the scheme and then, for comparison, to an otherwise identical enforcer without the scheme. Here, r=5 and the nodes' caches are enabled. Also, each stamp is used no more than twice; TESTs thus generate multiple GETs, some of which are dropped by the enforcer with-

out the scheme. Figure 10 graphs the positive responses as a function of the sent TEST rate. At high sent TEST rates, an enforcer with the scheme gives twice as many positive responses—that is, blocks more than twice as much spam—as an enforcer without the scheme.

6.7 Limitations

Although we have tested the enforcer under heavy load to verify that it does not degrade, we have not tested a flash crowd in which a *single* stamp s is GETed by all (several thousand) of the enforcer nodes. Note, however, that handling several thousand simultaneous GETs is not difficult because after a single disk seek for s, an assigned node has the needed key-value pair in its cache.

We have also not addressed heterogeneity. For *static* heterogeneity, *i.e.*, nodes that have unequal resources (*e.g.*, CPU, RAM), the bunker can adjust the load-balanced assignment of keys to values. *Dynamic* heterogeneity, *i.e.*, when certain nodes are busy, will be handled by the enforcer's robustness to unresponsive nodes and by the application's insensitivity to latency.

7 Related Work

We first place DQE in context with a survey of work on spam control (though space precludes a full list) and then compare the enforcer to related distributed systems.

7.1 Spam Control

Spam filters (e.g., [25, 59]) analyze incoming email to classify it as spam or legitimate. While these tools certainly offer inboxes much relief, they do not achieve our top-level goal of reliable email (see §1). Moreover, filters and spammers are in an arms race that makes classification ever harder.

The recently-proposed Re: [21] shares our reliable email goal. Re: uses friend-of-friend relationships to let correspondents whitelist each other automatically. In contrast to DQE, Re: allows *some* false positives (for non-whitelisted senders), but on the other hand does not require globally trusted entities (like the quota allocators and bunker, in our case). Templeton [62] proposes an infrastructure formed by cooperating ISPs to handle worldwide email; the infrastructure throttles email from untrusted sources that send too much. Like DQE, this proposal tries to control volumes but unlike DQE presumes the enforcement infrastructure is trusted. Other approaches include single-purpose addresses [31] and techniques by which email service providers can stop outbound spam [24].

In *postage* proposals (e.g., [20, 52]), senders pay receivers for each email; well-behaved receivers will not collect if the email is legitimate. This class of proposals is critiqued by Levine [38] and Abadi et al. [1]. Levine argues that creating a micropayment infrastructure to handle the world's email is infeasible and that potential

cheating is fatal. Abadi *et al.* argue that micropayments raise difficult issues because "protection against double spending [means] making currency vendor-specific There are numerous other issues ... when considering the use of a straight micro-commerce system. For example, sending email from your email account at your employer to your personal account at home would in effect steal money from your employer" [1].

With *pairwise* postage, receivers charge CPU cycles [4, 8, 17] or memory cycles [2, 16] (the latter being fairer because memory bandwidths are more uniform than CPU bandwidths) by asking senders to exhibit the solution of an appropriate puzzle. Similarly, receivers can demand human attention (*e.g.*, [60]) from a sender before reading an email.

Abadi et al. pioneered bankable postage [1]. Senders get tickets from a "Ticket Server" (TS) (perhaps by paying in memory cycles) and attach them to emails. Receivers check tickets for freshness, cancel them with the TS, and optionally refund them. Abadi et al. note that, compared to pairwise schemes, this approach offers: asynchrony (senders get tickets "off-line" without disrupting their workflow), stockpiling (senders can get tickets from various sources, e.g., their ISPs), and refunds (which conserve tickets when the receiver is friendly, giving a higher effective price to spammers, whose receivers would not refund).

DQE is a bankable postage scheme, but TS differs from DQE in three ways: first, it does not separate allocation and enforcement (see §2); second, it relies on a trusted central server; and third, it does not preserve sender-receiver email privacy. Another bankable postage scheme, SHRED [36], also has a central, trusted cancellation authority. Unlike TS and SHRED, DQE does not allow refunds (letting it do so is future work for us), though receivers can abstain from canceling stamps of known correspondents; see §8.1.

Goodmail [23]—now used by two major email providers [13]—resembles TS. (See also Bonded Sender [6], which is not a postage proposal but has the same goal as Goodmail.) Goodmail accredits bulk mailers, trying to ensure that they send only *solicited* email, and tags their email as "certified". The providers then bypass filters to deliver such email to their customers directly. However, Goodmail does not eliminate false positives because only "reputable bulk mailers" get this favored treatment. Moreover, like TS, Goodmail combines allocation and enforcement and does not preserve privacy.

7.2 Related Distributed Systems

Because the enforcer stores key-value pairs, DHTs seemed a natural substrate, and our first design used one. However, we abandoned them because (1) most DHTs

do not handle mutually untrusting nodes and (2) in most DHTs, nodes route requests for each other, which can decrease throughput if request handling is a bottleneck. Castro *et al.* [9] address (1) but use considerable mechanism to handle untrusting nodes that route requests for each other. Conversely, one-hop DHTs [28, 29] eschew routing, but nodes must trust each other to propagate membership information. In contrast, the enforcer relies on limited scale to avoid routing and on a trusted entity, the bunker (§4), to determine its membership.

Such static configuration is common; it is used by distributed systems that take the replicated state machine approach [55] to fault tolerance (*e.g.*, the Byzantine Fault Tolerant (BFT) literature [10], the recently proposed BAR model [3], and Rosebud [53]) as well as by Byzantine quorum solutions (*e.g.*, [40, 41]) and by cluster-based systems with strong semantics (*e.g.*, [26]).

What makes the enforcer unusual compared to the work just mentioned is that, to tolerate faults (Byzantine or otherwise), the enforcer does not need mechanism beyond the bunker: enforcer nodes do not need to know which other nodes are currently up (in contrast to replicated state machine solutions), and neither enforcer nodes nor enforcer clients try to protect data or ensure its consistency (in contrast to the Byzantine quorum literature and cluster-based systems with strong semantics). The reason the enforcer gets away with this simplicity is weak semantics. It stores only immutable data, and the entire application is robust to lost data.

8 Deployment and Economics

Though the following discussion is in the context of DQE, much of it applies to bankable postage [1] (or quota-based) proposals in general.

8.1 Deployment, Usage, Mailing Lists

Deployment We now speculate about paths to adoption. First, large email providers have an interest in reducing spam. A group of them could agree on a stamp format, allocate quotas to their users, and run the enforcer cooperatively. If each provider ran its own, separate enforcer, our design still applies: each enforcer must cope with a large universe of stamps. Another possibility is organization-by-organization adoption (the incremental benefit being that spoofed intra-organization spam no longer benefits from a "whitelist") or even individual-byindividual (the incremental benefit being that stamping one's email and sending to another DQE-enabled user ensures one's email will not be caught in a spam filter). In these cases, the deployment challenge is agreeing on a quota allocator and establishing an enforcer. The local changes (to email servers; email clients need not change) are less daunting.

Usage The amount of stamped spam will be negligible (see below for a rough argument). Thus, following the "no false positives" goal, stamped email should always be passed to the human user. For unstamped email: before DQE is widely deployed, this email should go through content filters (again risking false positives), and under widespread DQE deployment, this email can be considered spam. Conversely, DQE can incorporate whitelists, where people agree not to cancel the stamps of their frequent correspondents. Senders still stamp their mails to prevent spoofing, but such stamps do not "count" against the sender's quota. Such a social protocol is similar to TS's refunds [1].

Mailing lists For moderated lists, senders can spend a single stamp, and the list owner can then either sign the message or spend stamps for each receiver. Unmoderated, open mailing lists are problematic: spammers can multiply their effect while spending only one stamp. Partially-moderated lists might become more common under DOE. Here, messages from new contributors would be moderated (requiring only a glance to determine if the email is spam), and messages from known valid senders-based on past contributions and identified by the public key in the stamp—would be automatically sent to the list, again using either the list owner's public key or stamps for each recipient. In such lists, the moderation needed would be little (proportional to the number of messages from new contributors), so more lists could convert to this form.

8.2 Economics of Stamps

A quota allocation policy is effective whenever stamps cost a scarce resource. However, for simplicity, we view quotas as imposing a per-email monetary cost and do not discuss how to translate currencies like CPU [1], identity [5] or human attention [64] into money. Likewise, we only briefly consider how quotas should be allocated.

Basic analysis We give a rough argument about the effectiveness of a per-email cost. Assume that spammers are profit-maximizing and that, today, the industry (or individual spammers) make a maximal profit of P by sending m spam messages. Now assume that DQE is deployed and induces a stamp cost of c. Then, the maximum number of messages with fresh stamps that profit-maximizing spammers can send under DQE must be less than $\frac{P}{c}$: more would consume the entire maximal profit. To reduce spam (*i.e.*, m) by a factor f, one need only set $c = f\frac{P}{m}$. That is, to reduce spam by factor f, the price per message must be f times the profit-per-message.

The preceding analysis assumes that each stamp is reused only once, but adversaries can reuse each stamp a little more than once; see §4.1. Nevertheless, the analysis is very pessimistic: consider a variety of scams, each with a different profit-per-message when sent in the op-

timal amount. If, as we expect, most scams yield low profit, and few yield high profit, then setting a price c will prevent all scams with rate-of-return less than c. For example, if each scam sends the same amount, and if the number of scams returning more than a given amount q exponentially decays with q, then additive price increases in stamps result in multiplicative decreases in spam.

Pricing and allocation From the preceding analysis, the quota allocator should set a "price" according to a target reduction goal (f) and an estimate of spammer profits (P). Another option is for the quota allocator to monitor spam levels and find a price adaptively (though the feedback may occur on time scales that are too long). One problem is that, as argued by Laurie and Clayton in the context of computational puzzles [37], no price exists that affects spammers and not legitimate heavy users. In response, we note first that heavy users are the ones most affected by spam and might be willing to pay to reduce the problem. Second, the analysis in [37] does not take into account refunds (or uncanceled stamps, in our context), which, as Abadi et al. [1] point out, will strongly differentiate between a spammer (whose stamps will be canceled) and a legitimate heavy user.

A difficult policy question is: how can quota allocation give the poor fair sending rights without allowing spammers to send? We are not experts in this area and just mention one possibility. Perhaps a combination of explicit allocation in poor areas of the world, bundled quotas elsewhere (e.g., with an email account comes free stamps), and pricing for additional usage could impose the required price while making only heavy users pay.

9 Conclusion

The way DQE is supposed to work is that the economic mechanism of quotas will make stamps expensive for spammers while a technical mechanism—the enforcer—will keep stamps from "losing value" through too much reuse. Whether the first part of this supposition is wishful thinking is not a question we can answer, and our speculations about various policies and the future trajectory of email should be recognized as such. We are more confident, however, about the second part. Based on our work, we believe an enforcer that comprises a moderate number of dedicated, mutually untrusting hosts can handle stamp queries at the volume of the world's email. Such an infrastructure, together with the other technical mechanisms in DQE, meets the design goals in §2.

The enforcer's simplicity—particularly the minimal trust assumptions—encourages our belief in its practicality. Nevertheless, the enforcer was not an "easy problem." Its external structure, though now spare, is the end of a series of designs—and a few implementations—that we tried. By accepting that the bunker is a reasonable assumption and that lost data is not calamitous, we have

arrived at what we believe is a novel design point: a set of nodes that implement a simple storage abstraction but avoid neighbor maintenance, replica maintenance, and mutual trust. Moreover, the "price of distrust" in this system—in terms of what extra mechanisms are required because of mutual mistrust—is zero. We wonder whether this basic design would be useful in other contexts.

Appendix: Detailed Analysis

In this appendix, we justify the upper bound from $\S4.1$ on the expected number of uses of a stamp. We make a worst-case assumption that an adversary *tries* to reuse each stamp an infinite number of times. Observe that each use induces a PUT to an assigned node, and once the stamp is PUT to a good assigned node—good is defined in $\S4.1$ —the adversary can no longer reuse that stamp successfully. Since PUTs are random, some will be to a node that has already received a PUT for the stamp (in which case the node is bad), while others are to "new" nodes. Each time a PUT happens on a new node, there is a 1-p chance that the node is good.

$$E[I_{1}T_{1} + I_{2}T_{2} + \dots + I_{r}T_{r} + I_{r+1}T_{r+1}]$$

$$= E[I_{1}]E[T_{1}] + \dots + E[I_{r}]E[T_{r}] + E[I_{r+1}]E[T_{r+1}]$$

$$= 1 + p\frac{r}{r-1} + \dots + p^{r-1}\frac{r}{1} + p^{r}\left(n - \sum_{j=1}^{r} \frac{r}{r-j+1}\right)$$

$$= \sum_{i=0}^{r-1} p^{i}\frac{r}{r-i} + p^{r}\left(n - \sum_{j=1}^{r} \frac{r}{r-j+1}\right). \tag{1}$$

An upper bound for this expression is $\sum_{i=0}^{r-1} p^i \frac{r}{r-i} + p^r n$, which we can further bound by noting that $\frac{r}{r-i} \leq 2^i$ and assuming $p \leq 1/2$, giving an upper bound of

$$\frac{1}{1-2p} + p^r n. \tag{2}$$

Acknowledgments

We thank: Russ Cox, Dina Katabi, Sachin Katti, Sara Su, Arvind Thiagarajan, Mythili Vutukuru, and the anonymous reviewers, for their comments on drafts; David Andersen, Russ Cox, Sean Rhea, and Rodrigo

Rodrigues, for useful conversations; Russ Cox, Frank Dabek, Maxwell Krohn, and Emil Sit, for implementation suggestions; Shabsi Walfish, for many cryptography pointers; and Michel Goraczko and Emulab [18], for their invaluable help with experiments. This work was supported by the National Science Foundation under grants CNS-0225660 and CNS-0520241, by an NDSEG Graduate Fellowship, and by British Telecom.

Source code for the implementation described in §5 is available at:

http://nms.csail.mit.edu/dge

Notes

¹Although spam control is our motivating application, and certain details are specific to it, the general approach of issuing and canceling stamps can apply to any computational service (as noted before in [1]).

²Most of the ideas in §3.1 and §3.2 first appeared in [5].

³One might wonder why receivers will SET after they have already received "service" from the enforcer in the form of a TEST. Our answer is that executing these requests is inexpensive, automatic, and damaging to spammers.

⁴Our implementation uses SHA-1, which has recently been found to be weaker than previously thought [67]. We don't believe this weakness significantly affects our system because DQE stamps are valid for only two days, and, at least for the near future, any attack on SHA-1 is likely to require more computing resources than can be marshaled in this time. Moreover, DQE can easily move to another hash function.

⁵In this section (§6), we often use "stamp" to refer to the key-value pair associated with the stamp.

⁶We take n = 40(1 - p) instead of n = 40 because, as mentioned above, the Us issue TESTs and SETs only to the "up" nodes.

⁷The expression, with m = 40(1-p), is $(1-p)^3(1) + 3p^2(1-p)\alpha + 3p(1-p)^2\beta + p^3m\left(1-\left(\frac{m-1}{m}\right)^{32}\right)$. α is $\sum_{i=1}^m i\left(\frac{2}{3}\right)^{i-1}\frac{1}{m}\left(1+\frac{m-i}{3}\right)$, and β is $\sum_{i=1}^{m-1}i\left(\frac{1}{3}\right)^{i-1}\frac{m-i}{m(m-1)}\left(2+\frac{2}{3}\left(m-(i+1)\right)\right)$. See [66] for a derivation.

References

- M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In *Proc. Asian Computing Science Conference*, Dec. 2003.
- [2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In NDSS, 2003.
- [3] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In SOSP, Oct. 2005.
- [4] A. Back. Hashcash. http://www.cypherspace.org/adam/hashcash/.
- H. Balakrishnan and D. Karger. Spam-i-am: A proposal to combat spam using distributed quota management. In *HotNets*, Nov. 2004.
- [6] Bonded Sender Program. http://www.bondedsender.com/info_center.jsp.
- [7] Brightmail, Inc.: Spam percentages and spam categories. http://web.archive.org/web/20040811090755/http: //www.brightmail.com/spamstats.html.
- [8] Camram. http://www.camram.org/.
- [9] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In OSDI, Dec. 2002.
- [10] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM TOCS, 20(4):398–461, Nov. 2002.
- [11] ClickZ News. Costs of blocking legit e-mail to soar, Jan. 2004. http://www.clickz.com/news/article.php/3304671.

- [12] ClickZ News. Spam blocking experts: False positives inevitable, Feb. 2004.
 - http://www.clickz.com/news/article.php/3315541.
- [13] ClickZ News. AOL to implement e-mail certification program, Jan. 2006. http://www.clickz.com/news/article.php/3581301.
- [14] F. Dabek et al. Designing a DHT for low latency and high throughput. In NSDI, Mar. 2004.
- [15] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In OSDI, Oct. 1996.
- [16] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In CRYPTO, 2003.
- [17] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In CRYPTO, 1992.
- [18] Emulab. http://www.emulab.net.
- [19] Enterprise IT Planet. False positives: Spam's casualty of war costing billions, Aug. 2003. http://www.enterpriseitplanet.com/security/news/ article.php/2246371.
- [20] S. E. Fahlman. Selling interrupt rights: A way to control unwanted e-mail and telephone calls. *IBM Systems Journal*, 41(4):759–766, 2002.
- [21] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. Re: Reliable email. In NSDI, May 2006.
- [22] L. H. Gomes, C. Cazita, J. M. Almeida, V. Almeida, and W. Meira Jr. Charaterizing a spam traffic. In *IMC*, Oct. 2004.
- [23] Goodmail Systems. http://www.goodmailsystems.com.
- [24] J. Goodman and R. Rounthwaite. Stopping outgoing spam. In ACM Conf. on Electronic Commerce (EC), May 2004.
- [25] P. Graham. Better bayesian filtering. http://www.paulgraham.com/better.html.
- [26] S. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In OSDI, Oct. 2000.
- [27] R. F. Guilmette. ANNOUNCE: MONKEYS.COM: Now retired from spam fighting. newsgroup posting: news.admin.net-abuse.email, Sept. 2003.
- [28] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In NSDI, Mar. 2004.
- [29] I. Gupta, K. Birman, P. Linka, A. Demers, and R. van Renesse. Building an efficient and stable P2P DHT through increased memory and background overhead. In *IPTPS*, Feb. 2003.
- [30] IDC. Worldwide email usage forecast, 2005-2009: Email's future depends on keeping its value high and its cost low. http://www.idc.com/, Dec. 2005.
- [31] J. Ioannidis. Fighting spam by encapsulating policy in email addresses. In NDSS, 2003.
- [32] J. Jung and E. Sit. An empirical study of spam traffic and the use of DNS black lists. In IMC, Oct. 2004.
- [33] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In ACM STOC, May 1997.
- [34] D. E. Knuth. The Art of Computer Programming, chapter 6.4. Addison-Wesley, second edition, 1998.
- [35] H. Krawzyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, Feb. 1997. RFC 2104.
- [36] B. Krishnamurthy and E. Blackmond. SHRED: Spam harassment reduction via economic disincentives. http: //www.research.att.com/~bala/papers/shred-ext.ps, 2004.
- [37] B. Laurie and R. Clayton. "Proof-of-Work" proves not to work; version 0.2, Sept. 2004. http://www.cl.cam.ac.uk/users/rnc1/proofwork2.pdf.
- [38] J. R. Levine. An overview of e-postage. Taughannock Networks, http://www.taugh.com/epostage.pdf, 2003.
- [39] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In OSDI, Dec. 2004.

- [40] D. Malkhi and M. K. Reiter. Byzantine quorum systems. In ACM STOC, 1997.
- [41] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *IEEE Symp. on Reliable Distrib. Systems*, Oct. 1998.
- [42] D. Mazières. A toolkit for user-level file systems. In USENIX Technical Conference, June 2001.
- [43] MessageLabs Ltd. http://www.messagelabs.com/Threat_Watch/Threat_Statistics/Spam_Intercepts, 2006.
- [44] J. Mirkovic and P. Reiher. A taxonomy of DDoS attacks and DDoS defense mechanisms. CCR, 34(2), Apr. 2004.
- [45] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM TON*, 3(3):226–244, 1995.
- [46] PC World. Spam-proof your in-box, June 2004. http://www.pcworld.com/reviews/article/0, aid, 115885, 00. asp.
- [47] The Penny Black Project. http: //research.microsoft.com/research/sv/PennyBlack/.
- [48] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In OSDI, Dec. 2002.
- [49] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In USENIX FAST, Jan. 2002.
- [50] Radicati Group Inc.: Market Numbers Quarterly Update Q2 2003.
- [51] E. Ratliff. The zombie hunters. The New Yorker, Oct. 10 2005.
- [52] F.-R. Rideau. Stamps vs spam: Postage as a method to eliminate unsolicited commercial email. http: //fare.tunes.org/articles/stamps_vs_spam.html.
- [53] R. Rodrigues and B. Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003.
- [54] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. ACM TOCS, 10(1):26–52, 1992.
- [55] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, 22(4):299–319, Dec. 1990.
- [56] SecurityFocus. FBI busts alleged DDoS mafia, Aug. 2004. http://www.securityfocus.com/news/9411.
- [57] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In OSDI, Dec. 2002
- [58] M. Sherr, M. Greenwald, C. A. Gunter, S. Khanna, and S. S. Venkatesh. Mitigating DoS attack through selective bin verification. In 1st Wkshp. on Secure Netwk. Protcls., Nov. 2005.
- [59] SpamAssassin. http://spamassassin.apache.org/.
- [60] Spambouncer. http://www.spambouncer.org.
- [61] I. Stoica et al. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.
- [62] B. Templeton. Best way to end spam. http://www.templetons.com/brad/spam/endspam.html.
- [63] The Spamhaus Project. Spammers release virus to attack spamhaus.org. http://www.spamhaus.org/news.lasso?article=13, Nov. 2003.
- [64] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. CACM, 47(2), Feb. 2004.
- [65] M. Walfish, H. Balakrishnan, D. Karger, and S. Shenker. DoS: Fighting fire with fire. In *HotNets*, Nov. 2005.
- [66] M. Walfish, J. D. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Supplement to "Distributed Quota Enforcement for Spam Control". Technical report, MIT CSAIL, Apr. 2006. Available from http://nms.csail.mit.edu/dqe.
- [67] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In CRYPTO, Aug. 2005.
- [68] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In USENIX USITS, Mar. 2003.
- [69] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In SOSP, Oct. 2001.

RE: Reliable Email

Scott Garriss[†], Michael Kaminsky^{*} Michael J. Freedman[‡]°, Brad Karp^{**}, David Mazières°, Haifeng Yu^{*†} [†]Carnegie Mellon University, *Intel Research Pittsburgh, [‡]New York University, **University College London, °Stanford University

Abstract

The explosive growth in unwanted email has prompted the development of techniques for the *rejection* of email, intended to shield recipients from the onerous task of identifying the legitimate email in their inboxes amid a sea of spam. Unfortunately, widely used content-based filtering systems have converted the spam problem into a false positive one: email has become *unreliable*. Email *acceptance* techniques complement rejection ones; they can help prevent false positives by filing email into a user's inbox before it is considered for rejection. *Whitelisting*, whereby recipients accept email from some set of authorized senders, is one such acceptance technique.

We present Reliable Email (RE:), a new whitelisting system that incurs zero false positives among socially connected users. Unlike previous whitelisting systems, which require that whitelists be populated manually, RE: exploits *friend-of-friend* relationships among email correspondents to populate whitelists *automatically*. To do so, RE: permits an email's recipient to discover whether other email users have whitelisted the email's sender, while preserving the privacy of users' email contacts with cryptographic private matching techniques. Using real email traces from two sites, we demonstrate that RE: renders a significant fraction of received email reliable. Our evaluation also shows that RE: can prevent up to 88% of the false positives incurred by a widely deployed email rejection system, at modest computational cost.

1 Introduction and Motivation

Written communication is most useful when it is *reliable*; that is, when senders and recipients can both expect that a message sent will be received successfully by the intended recipient. Correspondents who are connected socially, whether directly or indirectly, often have something to lose if a message is not received (e.g., an urgent request from one colleague to another, or an urgent communication to a parent from her child's teacher). Similarly, a sender is most likely to expect a recipient to read and act on communication when the two parties have a preexisting social relationship, direct or indirect.

By the above definition, Internet email has been reliable throughout most of its entire long history, beginning with its 1971 origins on the ARPAnet. Email users have come to rely upon email reaching its destination. Today, however, unsolicited commercial email has rendered Internet email unreliable, as we explain below. We seek to restore email's reliability among correspondents who are linked to one another socially.

Spam inconveniences users by forcing them to search for legitimate email in an inbox dominated by chaff. Despite the efforts of legislative and law-enforcement bodies to the contrary, spam remains a pressing problem of large scale. In 2003, corporations spent an estimated \$2.5 billion on increased SMTP server capacity needed to process spam [30], and in July 2005, over 65% of email that crossed the Internet was spam [1]. The natural response of researchers and practitioners has been to develop and deploy a broad range of techniques intended to ensure that spam does not reach a user's inbox [2, 4, 6, 8, 19, 23, 29, 32, 33].

Content-based filtering has been particularly widely adopted as a spam defense strategy, perhaps because of its wide availability in free and commercial implementations. These systems are designed to reject email based on the presence of string tokens associated with spam. These tokens may either be selected manually by an administrator, or may be learned, most often by applying Bayesian learning to user-supplied training examples.

Many have reported great success at rejecting spam using content-based techniques, as measured by sub-1% false negative rates [20, 34]. Unfortunately, content-based filtering has replaced the spam problem with a false positive one. That is, misclassification of legitimate email as spam by content-based filters has rendered email unreliable. False positives are arguably more severe than spam in one's inbox, in that they are not merely a waste of a user's time—they represent possibly important email the user does not see.

The reasons why content-based filters result in false

¹SMTP server failures, disk exhaustion, and other hardware or software failures—or a recipient's simply not reading his email—can all of course result in delay or non-delivery. Such cases have not historically led users to view email itself as significantly unreliable, and some are not amenable to eradication by technical means. They are thus beyond the scope of consideration in this work.

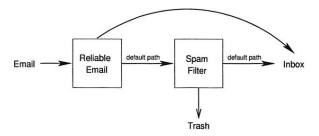


Figure 1: Complementary whitelisting and blacklisting.

positives are manifold.² A legitimate communication may contain strings associated with spam (e.g., "mortgage," "offer," "lottery," "Lagos," or any number of more colorful words that figure prominently in the descriptions of items flogged by spammers). Even if one were to build a content-based filter that could avoid false positives in these cases, others remain problematic, such as forwarding an interesting spam to a mailing list used for discussion by spam filter designers. In effect, content-based filters act as "dumb censors," as they prevent legitimate discussion on the basis of keyword matching—users can no longer say what they want in email.

In this paper, we propose Reliable Email (RE:), an automated email acceptance system based on whitelisting of email according to its sender. Figure 1 depicts a schematic view of how RE: fits into an email recipient's spam-fighting system. RE: is first to examine inbound email, and it delivers any message it accepts directly to the recipient's inbox. Note that RE: is entirely complementary to a mail rejection system; it cannot increase false positives because it either accepts a message or passes it to whatever rejection system was already in place.

The concept of a mail acceptance system is hardly new. Perhaps the simplest mail acceptance system is a *sender whitelist*, which places mail from senders enumerated on a list directly into the recipient's inbox. Today, whitelisting is rarely used in practice for three chief reasons:

- Whitelisting based on sender is trivially defeated by forging From: addresses, which are unauthenticated in SMTP. Even without knowing the contents of a recipient's whitelist, a spammer may trivially generate spam forged to appear to be *from* the recipient, whose address is most likely whitelisted.
- A recipient's whitelist cannot accept mail from a sender previously unknown to the recipient.
- Populating whitelists requires manual effort distributed diffusely in time, as users acquire new contacts.

RE: incorporates a mechanism to defeat forgery of From: addresses, as do other proposals that aim to stop spam [8, 33]. More significantly, RE: automatically broadens the set of senders whose mail is accepted by recipients' whitelists by explicitly examining the social network among email users. In particular, RE: allows user A to attest to user B. Such an attestation indicates that user A is willing to have email from user B directly filed in his inbox. An attestation thus roughly corresponds to the notion, "User A trusts user B not to send him spam." We say B is a friend of A. Clearly, attestations are useful for accepting mail in cases where a sender and recipient are friends; a sender may choose to generate an attestation for a recipient, and vice-versa, on the basis of the other party's identity.

We observe further that attestations are useful for accepting mail in cases where the sender and recipient are not already friends, but instead share a friend in common, a situation we term friend-of-friend (FoF). Suppose A and B are friends, B and C are friends, but A and C are as yet unknown to each other. If C sends email to A, the FoF relationship between A and C may give A confidence that C is not a spammer. That is, A may trust B not to be a spammer, B may trust C not to be a spammer, and on this basis, A may conclude that C is unlikely to be one.

Each email domain that participates in RE: runs a server that stores attestations. Together, the distributed collection of RE: servers allows *FoF queries* over attestations, whereby an email's recipient may determine whether the email's sender is an FoF. RE: thus allows a recipient to accept email from FoFs without requiring all users to trust a central authority.

Because attestations name email correspondents, allowing one user to query another user for attestations raises privacy concerns. RE: employs cryptographic private matching techniques to preserve the privacy of users' contact lists. Section 3.8 details the specific guarantees that RE: provides.

A central question is how useful FoF relationships are in increasing the number of emails RE: accepts into a user's inbox, versus the number of emails accepted by direct friend relationships alone. We consider this question in detail in Section 5. By way of motivation, we note briefly here that when evaluating the utility of social whitelisting using email traces from multiple sites, we find that RE: can accept almost 75% of received email and can prevent up to 88% of the false positives incurred by the existing spam filter. Moreover, augmenting friend relationships with FoF relationships increases the fraction of all received email accepted by RE: by at least 10%—a significant improvement in the fraction of received email rendered reliable.

We proceed in the remainder of this paper as follows. After reviewing related work in Section 2, Section 3 of-

²Some argue that spammers will eventually be able to evade Bayesian-trained content filters [10]; we are concerned chiefly with their false positives in this work.

fers design goals for a distributed whitelisting system and describes the design of RE: in detail. Section 4 describes our working RE: prototype, and Section 5 evaluates the system. Section 6 discusses various design decisions and open questions. We conclude in Section 7.

2 Related Work

We now survey the numerous and varied schemes proposed to fight spam, and compare these approaches to that taken in RE:.

Forgery Protection. Today's whitelists are often vulnerable to abuse because sender addresses are unauthenticated in SMTP, and thus may be forged trivially. Current methods for the prevention of mail forgery fall into two categories: digitally signed mail and trusted senders.

Digitally signing mail (e.g., with PGP [35]) allows recipients to authenticate the mail's content, including the sender's address. Clearly, requiring that all mail be digitally signed would solve the address forgery problem, but the use of digital signatures is hampered by the lack of any widely deployed public-key infrastructure.

Under trusted senders, a recipient can determine whether a received mail originated from a mail server in the domain of the sender's address. The Sender Policy Framework (SPF [33]), its derivative, Sender ID [26], and Yahoo! Domain Keys [8] exemplify this approach.

Social Networks. Ebel et al. [16] study the topology of an email social network and show that it exhibits small-world behavior. Kong et al. [23] use this finding to propose a collaborative, content-based email rejection system based on social networks, in which a user manually identifies the spam he receives, and publishes a digest of it to his social network. A user queries these digests to determine if mail he has received was previously classified as spam by others in his social network. This scheme presumes that users who are connected socially have the same definition of what constitutes spam *content*.

Ceglowski and Schachter [12] and Brickley and Miller [11] propose mechanisms for exchanging whitelist information using Bloom filters and SHA-1 hashes, respectively. These data structures do not contain cleartext whitelist entries, but are open to straightforward dictionary attacks. Both schemes assume sender addresses are not forged. Goldbeck and Hendler [19] present an algorithm that infers a trust score for a given sender based on social relationships, but compromises user privacy by requiring that users publish their social relationships.

PGP [35], though not originally intended to fight spam, uses a web-of-trust model for key distribution. The web-of-trust model relies on friend-of-friend trust relationships, as does RE:. RE:, however, uses these FoF relationships only to help identify legitimate senders, not for key distribution. RE: intentionally sidesteps the problem of robust key distribution, as described in Section 3.4.

Mail Rejection Systems. Machine learning techniques for text classification have been adapted to distinguish spam from legitimate email [21, 31]. Systems in this category, as exemplified by SpamAssassin [6], are by far the most widely deployed spam defense. As previously described, such systems reduce the reliability of email, as they inevitably classify some legitimate email as spam. Such false positives are a severe enough problem that some of the most accurate spam classification systems (e.g., [2]) resort to human labor to help with their classification, at greatly increased cost.

Since human-directed classification is accurate, but costly, several efforts have been made to distribute that cost across an email system's users. SpamNet [3], Distributed Checksum Clearinghouse (DCC) [4], and Razor2 [29] are all systems in which email users collaborate to classify spam. In each of these systems, users report spam to a centralized database. It is unclear how resilient these systems are to malicious users, particularly ones who mount a Sybil attack [13].

One may also reject email on the basis of the IP address of the sending host; past spam sources are thus blacklisted. Realtime Blackhole Lists (RBLs [5], or more generally, DNSBLs) and SpamCop [7] take this approach. These systems automate the distribution of a list of IP addresses known to have sent spam or run open SMTP relays. Blacklisting the SMTP server for a domain runs the risk of blacklisting many legitimate users. Because these lists are maintained in ad hoc fashion, it can be slow to have a server's reputation "cleared."

Other Mail Acceptance Systems. In proof-of-work schemes [9, 14, 15], a sender "pays" to send an email by solving a computational puzzle. Payment schemes rely on a similar notion, but they require senders to expend money rather than computation, e.g., by directly paying recipients [25]. For both classes of system, the intent is that the resources required to send mail will be prohibitively expensive in volume for a spammer, while affordable in smaller quantities for an average legitimate email sender. Laurie and Clayton [24] argue that spammers may harness the computational resources of large collections of compromised hosts to send spam in volume. Keeping email too expensive for spammers might then entail making it too expensive for heavy legitimate email users.

DQE [32] aims to limit the volume of email any one sender may send, without compromising the reliability of email between legitimate users. To achieve this goal, DQE employs a central authority trusted by all email

users, which allocates quotas of unforgeable stamps to email senders. Senders attach a stamp to every email they send. Recipients check the validity of the stamp on mail they receive. DQE eliminates false positives between sender-recipient pairs that adopt the system (and thus trust the same central quota allocator). When DQE is deployed incrementally, however, a recipient that wants protection from spam must fall back on a mail rejection system when it receives unstamped mail; in so doing, the recipient risks a false positive. We believe RE: and DQE complement one another well. RE: incurs no false positives for email between friends and FoFs, but cannot prevent false positives on mail whose sender is not a friend or FoF of the recipient. DQE, in contrast, can prevent false positives between any sender-recipient pair, provided they trust the same central quota allocator.

3 Design

We now continue by offering design goals for a robust and secure distributed email social whitelisting system. We then define terms useful in reasoning about RE:, describe RE:'s major components, and explain the system by way of a typical email usage scenario. We conclude this section by presenting the assumptions under which RE: operates and its corresponding security guarantees.

3.1 Design Goals

To be robust and secure, a distributed system that whitelists email and allows users to generate and query for attestations must provide three basic guarantees.

Sender addresses cannot be forged. Because a whitelisting system automatically accepts email based on sender address, protecting against this type of forgery is particularly important. Basic SMTP, however, does not provide any mechanism to prevent a user from sending email with an arbitrary From: address. RE: robustly verifies that the From: address in a received email has not been forged.

Attestations cannot be forged. An attestation is a statement by one user that another user is trusted to send email. Forging attestations would allow an adversary (spammer) to trick the recipient into accepting the adversary's email, even if the sender's address is not forged. RE: uses digital signatures to guarantee that attestations cannot be forged.

Privacy when exchanging whitelist information. A social whitelisting system can automatically accept email based on FoF relationships. A naive approach to learning about mutual friends is for one party to send his list of friends to the other party, who computes the set intersection. This approach, however, compromises the privacy

of the first party. RE: uses a private set-matching protocol to compute the intersection of sets of friends, but still provides provable privacy guarantees for both parties.

Additionally, any spam-fighting system is far more easily deployable if it provides two practical properties:

Incremental deployability. Inevitably, some users will adopt a spam-fighting system before others. A spam-fighting system ideally will offer real benefit to the community of users who have adopted it thus far. Because RE: is a mail acceptance system, it is complementary to pre-existing spam rejection systems, and thus easy to deploy incrementally. Clearly, the deployment of RE: does nothing to worsen the spam problem for those who have not adopted it. Moreover, RE: can confer benefit in the form of reduced false positives in any pairwise deployment, in which the domains of both sender and recipient run it.

Compatibility with today's SMTP. Making changes to the SMTP protocol slows adoption because developers must incorporate the new anti-spam SMTP protocol extensions into their mail software, and site administrators must then adopt one of the resulting new releases of this software. RE: does not change SMTP in any way; the system communicates over its own connections, and it is easily placed in the email processing path of most mail servers.

3.2 Definitions

The two key participants in any email exchange are the sender (S) and the recipient (R). Every user U in RE: possesses a public/secret key pair, denoted by PK_U and SK_U . The user keeps his secret key somewhere where he can access it while sending email. He registers his public key with his *Attestation Server* (AS), described below.

As described in Section 1, a user can issue an *attestation* to vouch that another user is a legitimate sender. An attestation by A for B, written $A \rightarrow B$, indicates that A trusts B to send email, and that A believes that other people should trust B to send email. Attestations contain the identities of the attester and the attestee, as well as an expiration time. More formally, an attestation is

$$A \rightarrow B = \{ \text{Hash}(A), \text{Hash}(B), start, duration} \}_{SK_A}$$

Hash() is a collision-resistant cryptographic hash (e.g., SHA-1) of the attester's or attestee's email address. The *start* and *duration* fields specify when the attestation expires. The entire attestation is cryptographically signed by the attester using his secret key.

Each SMTP domain that participates in RE: runs an Attestation Server (AS) responsible for storing attestations both by and to the users of that domain. The AS

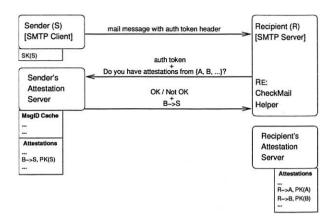


Figure 2: Sending Mail and Finding FoFs with RE:

also stores the public keys of its users as well as the public keys of users attested to by its users (i.e., for each attestation $A \rightarrow B$, where A is in the AS's domain, the AS stores PK_B). When the AS's users receive email from third-party senders, they use these attestee public keys to verify the signatures on the attestations that the third-party presents during an FoF query, as described below. The AS can run on the same machine as the domain's SMTP server, or on a different host, as indicated by DNS SRV records [22].

An AS responds to several types of client requests, such as queries for public keys, attestations, and FoFs. Some queries are restricted to users in the AS's domain (e.g., storing a public key) while others are open to everyone (e.g., getting a public key). Section 4 describes the full list of requests an AS supports.

3.3 Example: Sending Mail with RE:

Figure 2 shows an example of how two users running RE:, S and R, correspond by email. First, S composes a message to R and creates an authentication token (see Section 3.4). S signs the authentication token with his secret key and sends the message to R using standard SMTP. The authentication token is included as a header in the email message to avoid requiring modifications to the SMTP protocol.

R can decide to accept the mail based on the sender in one of two ways, depending on whether the sender is a direct friend or an FoF. These two cases differ in what R sends to S and what S returns. Figure 2 shows the FoF case.

To check for direct friendship, R examines the list of senders to whom he has attested (stored on R's AS); these attestations are of the form $R \to *$. If the recipient has attested to the sender (i.e., $R \to S$ exists), R contacts S's AS to verify the authentication token. In this case, the token verification is the only communication between R and S.

If the verification succeeds, RE: accepts the message and delivers it directly to the recipient's inbox.

If S is not one of R's direct friends, R will try to determine if S is an FoF. Specifically, R seeks to discover whether any of his direct friends has attested to S: is there an x such that $R \to x$ and $x \to S$? To answer this question, R queries the sender's AS. This query effectively contains a list of the recipient's direct friends. The sender's AS responds with a list of any attestations to the sender it holds from those direct friends. In Figure 2, the sender's AS returns $B \to S$, where B is a mutual friend. In this case, the communication between R and S combines the FoF query and verification of the authentication token.

In practice, *R* never actually sends its list of friends directly to *S*. Rather, *R* performs this FoF query using a private set matching protocol described in Section 3.8, which guarantees that the sender does not learn any of the recipient's direct friends and that the recipient only learns the attestations that are from his friends (i.e., the set intersection).

If R finds a match for this FoF query—e.g., an attestation to the sender by a B—it verifies the signature and freshness of this attestation. Note that R's own AS already has the mutual friend's public key, PK_B , stored alongside $R \rightarrow B$. If the attestation is valid and the authentication token verification succeeds, RE: accepts the message and delivers it to the recipient's inbox. Otherwise, R concludes that it is unable to determine whether or not the message is spam. R can then perform a default action (e.g., running the mail through a spam filter, graylisting it, etc.).

3.4 Sender Authentication

RE: attaches an *authentication token* to each outgoing email to protect against forged sender email addresses. This token is defined as follows:

 $\{Sender, Recipient, Timestamp, MessageID\}_{SK_S}$

The recipient verifies the authentication token before determining whether an incoming email should be whitelisted: (1) The recipient checks that the sender, recipient, and unique message ID match the values found in the message itself. This check ensures that the token cannot successfully be attached to an email destined to a different address. The recipient then connects to the sender's AS, which (2) checks that the authentication token is unused and (3) verifies the token's signature, as the AS knows the sender's public key PK_S . If an authentication token has been previously redeemed or the signature fails verification, the token check fails.³

³If after the recipient redeems the authentication token, one party suffers a crash or a network failure, the recipient will be unable to com-

Thus, authentication tokens serve two purposes. First, they allow recipients to reject forged sender addresses. Second, they prevent an adversary from making arbitrary FoF queries to the AS by replaying a previously redeemed token. Restricting FoF queries prevents the adversary from learning who has attested to the sender.

To verify that the authentication token is unused, the AS caches previously used tokens. To bound the size of this cache, the authentication token contains a timestamp. The AS only keeps tokens whose corresponding timestamps (times of issuance) are more recent than t seconds in the past. A reasonable value for t might be one week. Expiring authentication tokens based on timestamps assumes loosely synchronized clocks.

Because the recipient contacts the sender's AS to verify the authentication token, the recipient need not know the sender's public key. This method of verification assumes, however, that an adversary cannot perform a man-in-the-middle attack (see Section 3.7). Performing a man-in-the-middle attack in the wide-area network is non-trivial, and is far more work than a typical spammer would have the resources to perform for each email. Furthermore, successfully tricking the recipient into believing an invalid authentication token is valid results in, at worst, accepting a spam email into a user's inbox (i.e., a false negative).

3.5 Revocation

If a user's secret key is compromised or lost, authentication tokens and attestations signed by that key should be revoked. Once a user discovers that his key has been compromised, he uploads a new public key to his Attestation Server. From that point forward, the AS simply stops accepting authentication tokens signed with the old key.

For attestations, there are two cases that RE: must handle. First, when a recipient R's key is compromised, he should invalidate all attestations $R \to *$ stored at its own AS; we call these *local* attestations. When R uploads a new key to its AS, the AS can simply remove these local attestations; R can then re-issue the attestations as needed.

Second, R's friends must stop using R's attestations stored at their Attestation Servers; we call these *remote* attestations. Currently, RE: does not have a way to notify every one of R's friends automatically that his attestations are now invalid. RE: handles this case through expiration dates in attestations. Two expiration dates are

plete the SMTP transaction. When the sender retries, RE: will attempt to redeem the same authentication token a second time. This authentication check will fail, however, and RE: will not be able to accept the message automatically. In the current design, the recipient will then simply fall back to its default action. In future versions of the protocol, however, we intend to make RE: robust under such failures.

relevant: (1) The remote attestation itself (signed by R) has an expiration date and after this date, R's friends will stop using it. (2) During an FoF query, R's friends present these remote attestations to users who have attested to R, and these users attempt to verify the signature on the remote attestations using R's public key, which they stored when attesting to R. These users will reject the remote attestation if it is signed with an old, now invalid key. For example, if user U holds $U \to R$ and PK_R , U will re-fetch the attestation and PK_R once the attestation expires. Until then, however, U might accept mail from one of R's friends, or an adversary that can generate any $R \to *$ using R's compromised key.

A similar situation exists for revoking a single attestation, e.g., after a recipient *R* attests to a spammer by accident. To prevent the attestation's direct use (not FoF use), *R* simply removes the offending local attestation from his AS, thereby eliminating future false negatives from this spammer. This removal, however, does not help if the spammer (the attestee) is already holding a copy of the attestation, as he can subsequently use this remote attestation to send mail to users who have attested to *R*. Currently, RE: only limits the duration of a bad attestation by its expiration time, but alternatively the user can decide to remove his attestation for the recipient and thereby disregard the remote attestation presented by the spammer, as described in the following section.

3.6 Policy Decisions

RE: leaves several policy decisions to the user and/or system administrator. Most importantly, a user must decide when to create attestations. The most laborintensive but least error-prone policy is for the user to manually attest to other users after verifying that the sender is trusted to send email. This verification can be out-of-band (the user knows the sender personally) or can be based on the recommendation of a mutual friend learned through the RE: protocols. In the example above, the recipient *R* might decide to attest to the sender *S* because his mutual friend *B* attested to *S*. In this way, FoF queries help bootstrap a user's attestations to include a previously unknown correspondent.

Users can also specify policies that automatically create attestations. For example, a user might decide that anyone to whom he sends email is a trusted sender—presumably this user does not send email to spammers—and thus automatically create an attestation for each recipient of his outbound email. The user might also tune such a policy decision to include or exclude mail sent to particular domains. Another example of automatic attestation creation might be to attest to anyone that sends a user three non-spam emails that the user does not discard.

Users must also decide how to set expiration dates in an attestation. Expiration dates allow users to limit how long their friends will continue using the remote attestation. For example, a user might choose a more distant expiration date for personal acquaintances versus senders attested to automatically because they have sent three legitimate emails.

We presume that a user's trusted friend will only rarely attest to a spammer. If this friend does attest to a spammer, the user will accept mail from the spammer because of the FoF social relationship. The user will know, however, which friend attested to the spammer. In this sense, RE: limits the harm of attesting to a spammer: users can identify a friend's ill-considered remote attestation, and they can choose to ignore that friend's remote attestations when accepting inbound email in the future.

We elected to limit the social networking component of our system to one level (FoF). One might also consider using social relationships of three hops or longer in whitelisting email. We defer a discussion of this choice to Section 6.

3.7 Assumptions

In addition to standard cryptographic hardness assumptions, RE: operates under the following assumptions:

- Clocks are loosely synchronized to within some error bound. This is to ensure that (1) the sender's AS accepts only authentication tokens that are not too old, and that (2) the recipient uses only attestations that have not expired.
- An adversary cannot launch a man-in-the-middle attack. This assumption implies, for example, that the
 adversary cannot subvert forward DNS queries or
 intercept and modify IP packets traveling between
 an email sender and recipient.
- An adversary cannot compromise the sender's AS and/or convince it to lie about the validity of an authentication token or public key.
- An adversary cannot compromise a sender's machine. Section 6 discusses this assumption in more detail.

3.8 Privacy Protection

The FoF query allows the recipient to determine if any of his friends have attested to the sender. RE: performs this set intersection using a Private Matching (PM) protocol [18] that provides the following attractive privacy properties:

• The sender S does not learn anything about the recipient's friends. Both sender and recipient do learn

- an upper bound on the number of real friends presented by the opposite party, however.
- The recipient R learns only the intersection of the two sets of friends, i.e., those persons f for whom R→ f and f→ S. The PM protocol does not prevent parties from "lying" about their inputs; thus, the recipient can include arbitrary friends in his list when computing the set intersection. However, such inputs will fail later attestation verification and thus not result in a successful FoF chain.
- A third party observing all messages between sender and recipient learns only an upper bound on the size of each input, but nothing about their content nor the size of the intersection.
- No other party other than the recipient can execute the FoF query, as the AS only allows one query per valid authentication token.

RE:'s private matching protocol is a type of secure two-party computation that is optimized for computation and communication efficiency. At a high level, the basic protocol has three steps:

- The recipient encodes his k_R friends' names in a special encrypted data structure, which he sends to the sender.
- 2. The sender performs a computation on the encrypted data structure with each of her k_S friends' names and the corresponding attestations, generating k_S outputs. If the sender's friend f is encoded within the encrypted data structure, an output's underlying plaintext becomes the attestation f → S; otherwise, that output's plaintext becomes random. She sends the k_S outputs back to the recipient.
- The recipient decrypts these k_S results. He recovers the attestations corresponding to their set of friends in common.

Our private matching protocol takes advantage of the special mathematical properties of certain public-key encryption schemes, such as Paillier [28] and a variant of ElGamal [17]⁴, that preserve the group homomorphism of addition and allow multiplication by a constant. In other words, the following operations can be performed without knowledge of the private key: (1) Given two encryptions $enc(m_1)$ and $enc(m_2)$, one can compute $enc(m_1+m_2)=enc(m_1)\cdot enc(m_2)$. (2) Given some constant c belonging to the same group, $enc(cm)=enc(m)^c$.

⁴Standard ElGamal, which encrypts a message m as $\langle g^x, g^{xr}m \rangle$, does not directly support the necessary homomorphic operations required. Thus, when requiring these homomorphic properties, we encrypt m as $\langle g^x, g^{xr}g^m \rangle$, even though decrypting this ciphertext only recovers g^m , not m! This variant is sufficient for almost all operations, with the exception of the sender encrypting its attestation for later recovery by the recipient, which therefore uses standard ElGamal.

We use the following corollary of these properties: (3) Given encryptions of the coefficients a_0, \ldots, a_k of a polynomial P of degree k, and a plaintext y, one can compute an encryption of P(y).

We now construct a basic secure PM protocol in the following manner:

 The recipient R defines a polynomial P whose roots are hash values encoding his k_R friends, i.e., given x_i ← Hash(f_i^R), R computes:

$$P(y) = (x_1 - y)(x_2 - y) \dots (x_{k_R} - y) = \sum_{u=0}^{k_R} a_u y^u$$

R encrypts these k_R coefficients under his public key and sends the resulting ciphertexts to S.

2. The sender S uses the homomorphic properties of the encryption system to evaluate the polynomial on each hash value of her k_S friends, i.e., $\forall i, y_i \leftarrow \operatorname{Hash}(f_i^S)$:

$$enc(P(y_i)) = enc(a_0) \left(enc(a_1) \left(\dots enc(a_{k_R})^{y_i}\right)^{y_i}\right)^{y_i}$$

S then multiplies each $P(y_i)$ result by a fresh random number r to get an intermediate result, and she adds it to her corresponding attestation from f_i^S (encrypted under R's public key):

$$enc\left(r\cdot P(y_i) + \{f_i^S \to S\}\right)$$

S randomly permutes this set and returns it to R.

3. R decrypts each element of this set. For every friend in common, P(y_i) = 0 and R recovers the attestation f_i^S → S. Otherwise, P(y_i) is non-zero and the resulting decryption appears random. R checks that f_i^S is in its friends list and verifies the attestation f_i^S → S before accepting the FoF chain.

For proofs of the protocol's security, as well as efficiency optimizations and other implementation details, we refer the reader to Freedman et al. [18].

Such cryptographic tools must be applied with care to prevent information leakage that unknowingly introduces side-channel attacks against the protocol's privacy. For example, one may be tempted to distribute public keys inside FoF attestations, much like a web-of-trust for key distribution. However, this approach can be used to break the system's privacy. Consider the case in which a sender maintains multiple public keys. If the sender receives attestations from friends on different keys, and a recipient accepts one such key following a successful FoF protocol with the sender and then subsequently attests to it, the sender can immediately deduce which friend the two parties have in common. Thus, all identifying information (public keys, email addresses) must be retrieved directly from their corresponding parties, in order to ensure consistency across participants and thus prevent such side-channel attacks.

Public RPCs	Private RPCs		
GetPK	GetWhitelist		
SubmitAtt	GetAtt		
CheckAuth FindFriend	SetPK		

Table 1: RPC interface to the Attestation Server

4 Implementation

The current RE: prototype consists of the Attestation Server, Private Matching (PM) implementation, and a number of utilities for creating attestations, authentication tokens, and for checking incoming mail. The prototype is built atop the SFS toolkit [27], which provides primitives for asynchronous event-driven programming, cryptography, and RPC. We have integrated the RE: prototype with the Mutt mail client, the Mail Avenger SMTP server, and the Postfix SMTP client. Note that we made no modifications to any of these pre-existing mail tools. RE: is roughly 4500 lines of C++ source, plus 275 lines of Sun XDR code for the RE: wire protocol specification. RE: uses DSA for digital signatures on attestations and authentication tokens. For the PM protocol, RE: uses the ElGamal variant [17].

4.1 Attestation Server

Currently, each email domain that wishes to participate in RE: must run an Attestation Server. The domain publishes SRV records [22] to indicate the location of the AS. For each user *A* in the domain, the Attestation Server is responsible for maintaining the following:

- A's public key PKA
- all unexpired, redeemed authentication tokens generated by A when sending email
- all attestations A → x created by A, along with x's public key PK_x
- all remote attestations $x \rightarrow A$ for A

In addition to maintaining this information, the Attestation Server provides two RPC interfaces (see Table 1). The public interface is used by email recipients to decide whether to accept mail sent from this domain. The private interface, requiring authentication, is used only by the domain's recipients: when processing incoming mail, for updating public keys, and for providing users access to their attestations and the corresponding attestee public keys. The current implementation restricts access to the private RPCs to clients running on the same machine as the AS; future versions of the Attestation Server could allow clients to authenticate through existing security mechanisms (e.g., SSL, Kerberos) and/or provide a

secure Web interface to the private RPCs (e.g., SetPK). We now describe the public RPCs in detail.

GetPK retrieves the public key for a user in this domain. This function is used only when a user generates an attestation *for* a user in this domain.

SubmitAtt serves two purposes. The first is to allow users in this domain to upload attestations that they have created for other users. The Attestation Server knows the public keys for its local users, and can thus verify that an attestation is valid before accepting it. The second purpose of SubmitAtt is to accept attestations created by other users for a user in this domain. In this situation, the Attestation Server does not know the public key of the issuer of the attestation, so it stores the attestation without verifying it. Attestations of this type are only used as part of a transitive link, and they are verified by the person who wishes to use the transitive link to accept mail.

CheckAuth verifies the authentication tokens included in messages sent by users in this domain. The AS verifies the token's digital signature and that the authentication token has not been used previously. To enforce the latter, the server caches tokens that have been redeemed, but it limits the size of the cache by rejecting tokens older than a fixed window of time (e.g., one week).

FindFriend uses PM to compute the intersection between the supplied set of email addresses and the email addresses of people who have attested to the sender. The authentication token is included as a parameter to FindFriend so that the recipient does not need to issue a separate CheckAuth query.

4.2 Incremental Deployment

One requirement of any practical system is that it be incrementally deployable. Since RE: either accepts an email on the basis of attestations or passes the email to the user's traditional spam filter, deploying RE: does not adversely affect a user's ability to send and receive mail.

The current RE: prototype is deployable on a perdomain basis. A domain that wishes to use RE: must install and configure an Attestation Server plus make various RE: utilities available to its users. In the future, we envision that these utilities will exist as easily downloadable plugins for popular mail clients. To ease early adoption, future versions of RE: may also allow users to specify an alternative AS for situations in which the user's ISP does not run its own AS.

It is important to note that RE: is also incrementally deployable *within* domains themselves. Users who do not download and install the new client software will still be able to send and receive mail; they merely will not be able to take advantage of the reliability offered by RE:.

5 Evaluation

We now turn to evaluating RE:; particularly, we are concerned with its effectiveness at whitelisting email, its potential to reduce false positives, the computational cost of private matching in the system, and the end-to-end email processing throughput attainable when a standard email processing system is augmented with RE:.

5.1 Effectiveness of RE:

The goal of RE: is to accept email, thereby preventing these messages from becoming false positives in a mail rejection system. To ascertain the effectiveness of RE: at meeting this goal, our evaluation must answer three questions:

- Overall utility: What fraction of inbound emails are accepted by RE:? This fraction of email is protected from becoming false positives.
- 2. **Utility of FoF relationships:** By how much do FoF relationships improve RE:'s effectiveness in accepting email beyond direct attestations?
- 3. End-to-end effectiveness: If RE: were deployed, would it eliminate real false positives produced by today's rejection systems?

Ideally, we would perform a controlled experiment. We would find a large population of email users and have them run both RE: and a traditional content-based email rejection system, side-by-side. We would pass a copy of every inbound email to each of the two systems. For RE:, users would create attestations as they saw fit, typically upon sending or receiving email. As the social network (distributedly stored attestations) builds up, we would measure the fraction of accepted email. For the rejection system, we would have users inspect a spam folder that contains messages that the rejection system flagged as spam, and manually note any false positives. Finally, we would directly count the emails accepted by RE: that would have been false positives had the rejection system been deployed alone.

Unfortunately, we do not have a large-scale deployment. We do, however, have email traces from a large corporation and a large university. For the corporate dataset, we also have data on false positives reported to the email system administrators. Using this information, we can approximate the ideal experiments. The main problem, however, is that we must emulate user generation of attestations, as described below.

Email traces. The first trace is taken from a large corporation with well over 80,000 employees, all of whom use email. The trace covers about one month of email and contains over 63 million anonymized messages, where a

message with multiple recipients counts as one message per recipient. Because of the corporation's email server architecture and the placement of the monitoring point, email in the trace is nearly exclusively email sent into and out of the corporate network (not mail internal to the network). The corporation's mail server uses a popular content-based spam filter, and the trace indicates whether or not the server's spam filter judged each message to be spam (via a spam score).

The second trace covers New York University's toplevel domain, which includes approximately 60,000 email users. The trace covers about one week of email and contains approximately 6 million anonymized messages. NYU uses the SpamAssassin-derived PureMessage filter, and the trace also contains a spam score for each message. Unlike the corporate trace, this university trace also contains messages sent within the domain (e.g., between two students).

A model for creating attestations. Lacking the social network for the users in our email traces, we developed a model for when users might create attestations based on the trace itself and the following assumptions. These rules were applied while processing the traces in chronological order, ignoring all messages that were flagged as spam by the server's content-filter.

- 1. Every sender attests to every recipient.
- Every recipient attests to every sender.
- (Corporate trace only) Every employee attests to every other employee.

We believe that the first rule is reasonable since we assume that the messages considered are not spam. Presumably, users are not sending email to spammers; furthermore, by attesting to the recipient, they will automatically whitelist any replies. The second rule would be dangerous, of course, if the trace contained spam, as the user would whitelist the spammer. Again, we assume that all messages considered are not spam. Even so, this rule might be optimistic, but it is useful for bootstrapping the social network from a trace. The second rule means that the recipient will accept all subsequent mail from that sender (the first message, though, would not be whitelisted as the rule is applied after processing the message). The third rule is specific to the corporate trace; the rule's purpose is to capture the large body of email messages that do not appear in the trace—the messages between users within the corporate email network. In a real deployment, these messages and the resulting attestations would provide critical links in the social network.

The evaluation of the corporate dataset contains two sources of error that result directly from limitations of the dataset itself. The first source of error is the attestation model described above, which assumes that every employee at the company attests to every other employee.

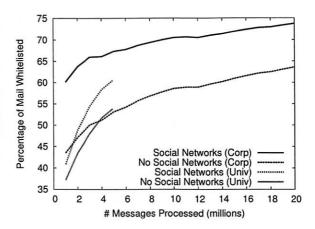


Figure 3: RE: Total coverage

This assumption may cause us to overestimate the hit rate for whitelisting. A more conservative rule for generating intra-company attestations might produce a lower hit rate. The second source of error is that the dataset does not include intra-company messages. This omission may cause us to underestimate the hit rate for whitelisting.

5.1.1 Fraction of Mail Whitelisted

Given these traces and a model for when users would create attestations, our first experiment was to measure the fraction of mail that RE: could whitelist based on social relationships (both direct and FoF). RE:'s ability to whitelist messages gives an indication of what fraction of mail the system can make reliable.

We consider two measures of whitelisting's efficacy. First, we examine total whitelisting coverage, or the fraction of all emails whitelisted. Second, we examine whitelisting coverage for email from *strangers*. When there is no previous message from a sender to a recipient, we term the sender a stranger. It is precisely in this case where FoF relationships are crucial to the acceptance of email.

Figure 3 shows the results of the total coverage experiment. The *x*-axis shows the number of email messages processed (in chronological order) so far, and the *y*-axis shows the percentage of email received so far (i.e., percentage of *x*) that the system was able to whitelist. The attestations are created as the trace is processed. The lower curve (for each trace) is the effectiveness of whitelisting, based on our attestation generation rules above, without considering social links. The upper curve plots the additional effectiveness of the whitelist when one accepts mail from FoFs.

The two traces are plotted on the same graph to ease comparison. Examining the 5 million email mark, where the university trace ends, we see that the two environments have relatively similar results. In the corporate trace, after processing 20 million non-spam mes-

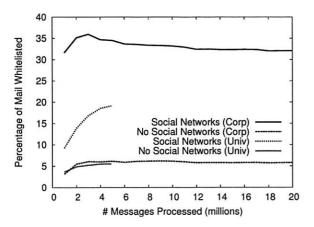


Figure 4: RE: Stranger coverage

sages, social whitelisting can accept almost 75% of email received, approximately 10% more than the direct whitelist. In the one-week university trace, after processing almost 5 million non-spam messages, social whitelisting can accept over 60% of email received, approximately 7% more than direct whitelisting.

Figure 4 shows the results of the strangers coverage experiment, with the x-axis as before, and the y-axis depicting the fraction of email received from strangers that the system was able to whitelist. While the attestation generation rules used in this experiment are the same as before, direct attestation whitelists very few emails, as one expects: only emails from never-before-heard-from senders are considered. In fact, the only emails whitelisted by direct attestation are the first replies sent by strangers in response to emails from recipients. FoF queries are able to whitelist an additional 26% (corporate) or 13% (university) of email from strangers.

5.1.2 False Positives Saved

To make email reliable, RE: must reduce the incidence of false positives. Specifically, RE: can help by accepting email messages from friends or FoFs that would otherwise be classified as spam. As noted above, we were able to obtain false positive data for the corporate environment but not for the university one.

The corporate false positive data is a list of all userreported false positives during the one-month trace period. Typically, these reports are submitted by users outside of the company who received a bounce message from the company's mail server after the spam filter (incorrectly) determined the user's message was spam. The list contains the anonymized sender and recipient email addresses for each of these reports.

Using this data, we measured the effectiveness of RE: at reducing false positives as follows. We ran the same experiment described in Section 5.1.1, except we did not automatically drop all messages that were flagged as

	Encrypt	Decrypt	HAdd
Paillier	21.8	1.5	.017
ElGamal	1.7	0.8	.010

Table 2: Speed (ms) of 1024-bit public-key operations

spam in the trace. Instead, for each of these flagged messages, we consulted the list of reported false positives to see if it contained the message's sender-recipient pair. If so, we assumed that the message was not spam, but rather a false positive. This assumption is reasonable if one assumes that spammers did not report false positives to the company in order to get their spam through and that spammers did not forge mail with the exact same sender-recipient pair that appeared in the false positive list.

For each false positive identified during the trace, we recorded if the system would accept that message based on the current social network. Out of 20 million messages, we identified 172 false positives. Of those false positives, approximately 84% would have been whitelisted by RE: using direct attestations and an additional 5% using FoF relationships. We note that the number of reported false positives is most likely much lower than the number of actual false positives.

5.2 Microbenchmarks

This section provides microbenchmarks for the speed of homomorphic cryptosystems and private matching protocol [18] implementations. Benchmarks were performed on a 3 GHz Intel® Xeon® processor-based computer running in 64-bit mode.

Table 2 shows the performance of Paillier [28] (in fast decryption mode) and ElGamal [17] operations given as the mean over 300 runs across five different keys. HAdd corresponds to the cost of performing a homomorphic addition of plaintexts, which is akin to modular multiplication. Due to its superior performance, we use ElGamal to instantiate our PM protocol. This section's subsequent PM benchmarks reflect this choice of cryptosystem.

There are three stages to evaluate in the Private Matching protocol: (1) the recipient's setup time to construct an encrypted polynomial, (2) the sender's evaluation of this polynomial on its inputs, and (3) the recipient's subsequent recovery of the intersection.

The recipient's setup time is directly measured by k_R encryptions. These encryptions only need to be precomputed *once* per input set (friendship list); the recipient recomputes these encrypted coefficients upon adding or removing an element from its input.

Figure 5 shows the sender-side performance of PM, given as the mean of three runs. We graph performance for varying sender (k_S) and recipient (k_R) input sizes. Re-

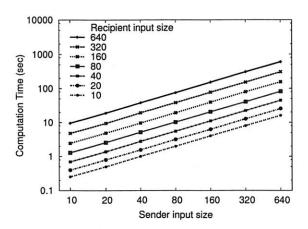


Figure 5: Sender-side PM performance (seconds).

S input size	10	20	40	80	160
R comp. time (s)	.008	.015	.032	.063	.126

Table 3: Time (s) for recipient to recover intersection

call that the sender must perform $O(k_S \cdot k_R)$ operations, i.e., to evaluate each of its inputs on a degree- k_R polynomial. Yet, we see the practical effect of using Horner's Rule for evaluating the polynomial "from the inside out": most homomorphic operations work with exponents of small size. Thus, *in practice*, the performance appears to grow only linearly in both k_R and k_S for reasonably small input sizes (i.e., its slope in log-log scale is 1).

The sender-side computation overhead is acceptable for smaller set sizes. For example, when a sender and recipient each have a set of 40 friends they want to intersect, the sender spends 2.8 seconds to complete the PM computation. At larger sizes, the overhead is more noticeable. To intersect two sets of 160 friends, for example, the sender needs almost 40 seconds of computation.

We make several observations as to this performance: (1) The asymptotic running time of the PM can be reduced to $O(k_S(1 + \ln \ln k_R))$ by using multiple lowdegree polynomials for potential performance improvements [18]. (2) All operations are completely parallelizable onto multi-processor architectures, as each computation on the sender's k_S inputs are independent of one another. Furthermore, a single domain can deploy multiple Attestation Servers to distribute the load. Given the money that people spend on fighting spam today, we believe the additional cost is reasonable. (3) Alternatively, the cryptographic operations can be partially or fully implemented in hardware for higher performance. (4) Finally, we observe from Section 5.1.1 that a large fraction of messages for certain RE: deployments may be satisfied with direct attestations and thus would never perform FoF queries in the first place.

Last, we examine the recipient's time to recover the

	Mean	StdDev
Mail Avenger	16.7	0.57
RE: + Mail Avenger	14.4	0.76

Table 4: Delivery throughput (messages/second)

intersection (see Table 3). Note that performance is independent of the recipient's input and computation can again be parallelized over k_S processors. We also conclude that most of the protocol's load is placed on the sender, which provides some protection against computational DoS attacks against email recipients.

5.3 Throughput

To analyze end-to-end throughput, we augment a standard mail processing system with RE: and measure its impact. For all experiments, the SMTP server is a 2 GHz Intel® Pentium® 4 processor-based computer running Mail Avenger, and all machines are connected via a local area network.

First, we measure how RE: affects the rate at which mail is received by measuring how much time an SMTP server needs to process mail with and without RE:. Our experimental setup consists of our SMTP server and three senders, each running on a different machine. The senders simultaneously bombard the SMTP server with sufficiently many messages to saturate it (e.g., sending 500 messages each as rapidly as possible). In the RE: case, the recipient has attested to all of the senders, and each sender serves as its own AS for authentication checks. Table 4 shows that the addition of RE: reduces throughput by 13.5%.

Second, we measure the time it takes for an AS to process CheckAuth and FindFriend queries from recipients. Because the AS must be contacted for each outgoing mail, this time affects the rate at which a domain's users can send mail. (Note that the processing time required on the client to add an authentication token to each message is the time to generate a digital signature times the number of recipients.) To measure the query-processing rate on the AS, we have a single sender periodically send a mail to our SMTP server. For these tests, the sender's AS is a 3 GHz Intel® Xeon® processor-based computer running in 64-bit mode. The average time to process CheckAuth and FindFriend⁵ queries is 0.0516 and 2.85 seconds, respectively. Equivalently, the AS can process 1162 CheckAuth queries per minute or 21.1 FindFriend queries per minute. We note here too the performance observations enumerated in the previous section, which apply to the FindFriend queries.

⁵PM protocol performed using 40 friends in each set.

6 Discussion

We now discuss various design decisions, open questions, and useful enhancements to RE:.

Compromised Senders and Audit Trails. Under normal operation, RE: assumes that a sender's machine is not compromised (see Section 3.7). If this assumption is violated, a spammer's message will appear to come from a legitimate user. Unfortunately, any acceptance system based on identity (e.g., the sender) or even an opaque token (e.g., a stamp) is susceptible to attacks in which the sender is compromised. The problem of compromised senders, though, is just about false negatives: at worst, a compromised sender in RE: can cause a recipient to accept spam.

RE: provides the nice property, however, that the recipient knows with some degree of certainty—due to the authentication token—who sent the received spam and why RE: accepted it. The "why" can be either because of a direct attestation or an FoF relationship. For example, this audit trail might indicate that RE: accepted the spam because the recipient attested to Bob and Bob attested to the sender. Given this information, a recipient can choose to reconsider using his attestations for Bob.

False Negatives. Unfortunately, our anonymized traces do not provide sufficient information to quantify how RE: would affect the false negative rate. Qualitatively, however, RE: is robust against false negatives: a false negative can occur in RE: only if the recipient has attested directly to a spammer or is connected via an FoF chain. In either case, the recipient knows who the responsible party is, providing an important audit trail that describes the trust relationship between recipient and sender.

An attestation for a spammer might exist for several reasons. A user may erroneously attest, he may have been compromised by a virus, or he may be malicious and attest to a spammer intentionally. One simple way to limit the damage caused by machine compromises is to require a password to generate a new attestation.⁶

Mail with Multiple Recipients. Our previous discussion has been confined to mail that is addressed to a single recipient. Many messages, however, contain multiple recipients. These recipients are either listed explicitly, or they are hidden behind a mailing list. In the former case, the sender must generate a different authentication token for each recipient because the attestation server rejects duplicate tokens to prevent replay attacks (i.e., the AS ensures one FoF query per token).

The sender must make sure, however, to send each token to only the corresponding recipient. Sending all of the tokens to all of the recipients would allow one misbehaving recipient to use up all of the tokens to make FoF queries to the AS. Other recipients would get an error when trying to use their tokens because the request would look like a replay attack. Note that the AS has no way to ensure that the recipient is using the "correct" authentication token (the one in which she appears) because the AS has no way to identify the requester as any particular recipient.

Regarding mailing lists, the simplest solution is for users to attest to the mailing list itself. The mailing list moderator is then responsible for ensuring that spam does not make it on to the list. Unfortunately, unmoderated lists still pose a problem—there is no simple solution beyond the current content-based filtering.

Sender Privacy and Profiles. One inherent property of the RE: design is that senders do reveal a subset of their friends to the recipient (i.e., those that intersect with the recipient's list of friends). Thus, a malicious recipient has the opportunity to query the sender with an arbitrary list of friends to discover who has attested to the sender. We can address this issue partially with sender profiles, which can allow the sender to control what attestations the sender uses during the FoF query. A sender might choose to have a profile with a restricted set of attestations-i.e., maintaining separate personal and work profiles-so the recipient cannot learn about specific people who have attested to the sender. The potential downside of eliminating attestations from a profile is that the chance of intersecting with the recipient's set of friends, and thus the sender having his mail automatically accepted, is lower.

Length of Social Paths. In this work, we have only explored direct and FoF social relationships, but clearly longer paths bear examination: they are more inclusive of previously unknown senders, and could thus potentially whitelist more email. On the other hand, there is also a tension between this greater coverage and an increased risk of false negatives. As the length of the shortest social path between the sender and recipient increases, it becomes increasingly unclear whether the recipient can safely automatically accept email. If and how private matching protocols would work beyond two degrees of separation is also an open question; furthermore, even if providing privacy with longer chains of trust were possible, it would likely require that other parties be online during the mail transaction besides the sender and recipient.

7 Conclusion

Motivated by the decline in the end-to-end reliability of email at the hands of spam-rejection systems, we have described RE:, a system for automatically accept-

⁶Of course, a sophisticated adversary could use a keystroke sniffer.

ing mail based on its sender. As an acceptance system, RE: is complementary to existing spam-defense systems; it simply bypasses mail rejection systems for senders who are deemed trustworthy. RE: improves upon standard whitelisting approaches in two ways: by preventing sender forgery though the use of an authentication token, and more importantly, by increasing the fraction of email that can be whitelisted through the examination of social relationships, while preserving the privacy of users' correspondents.

We have shown that RE: can reliably deliver the majority of a site's incoming mail; furthermore, augmenting direct-friend attestations with friend-of-friend relationships significantly increases the percentage of accepted mail. More importantly, experiments show that RE: can eliminate a large percentage of false positives produced by an existing content-based spam filter. Our full implementation of RE: does not significantly reduce the rate at which an SMTP server can accept incoming mail or impose a substantial bottleneck on the rate at which users of a domain can send mail.

8 Acknowledgments

The authors thank the anonymous reviewers, their shepherd Dan Rubenstein, Michael Walfish, and Mark Handley for their feedback and comments. The authors also extend a special thanks to Michael Puskar and NYU ITS, Robert Johnson, Marc Foster, and Greg Matthews, without whom the data analysis in this paper would not have been possible. Antonio Nicolosi and Benny Pinkas provided valuable help with the cryptography, and Michael Ryan contributed to the code. This work was partially supported by project IRIS under NSF Cooperative Agreement ANI-0225660.

References

- MessageLabs intelligence report: Spam intercepts timeline, July 2005. http://www.messagelabs.co.uk/.
- [2] Symantec Brightmail AntiSpam. http://www.brightmail.com/.
- [3] CloudMark. http://www.cloudmark.com/.
- [4] Distributed checksum clearinghouse, Oct. 2005. http://www.rhyolite.com/anti-spam/dcc/dcc-tree/dcc.html.
- [5] TrendMicro's RBL+. http://www.mail-abuse.com/.
- [6] SpamAssassin, . http://spamassassin.apache.org/.
- [7] SpamCop, http://www.spamcop.net/.
- [8] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. Domainkeys identified mail (DKIM). Internet Engineering Task Force (IETF) Draft, July 2005.
- [9] A. Back. Hashcash, May 1997. http://www.cypherspace.org/hashcash/.
- [10] T. Blackwell. Why spam cannot be stopped, June 2004. http: //tlb.org/whyspamcannotbestopped.html.

- [11] D. Brickley and L. Miller. FOAF, 2005. http://xmlns. com/foaf/0.1.
- [12] M. Ceglowski and J. Schachter. LOAF, 2004. http://loaf.cantbedone.org/.
- [13] J. R. Douceur. The sybil attack. In First International Workshop on Peer-to-Peer Systems (IPTPS), Mar. 2002.
- [14] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In Advances in Cryptology (CRYPTO), volume 740 of Lecture Notes in Computer Science, 1992.
- [15] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In Advances in Cryptology (CRYPTO), volume 2729 of Lecture Notes in Computer Science, 2003.
- [16] H. Ebel, L.-I. Mielsch, and S. Bornholdt. Scale-free topology of e-mail networks. In *Physical Review E* 66, 2002.
- [17] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
- [18] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In Advances in Cryptology — EU-ROCRYPT 2004, May 2004.
- [19] J. Golbeck and J. Hendler. Reputation network analysis for email filtering. In Conference on Email and Anti-Spam (CEAS), 2004.
- [20] P. Graham. Better bayesian filtering. In MIT Spam Conference, Jan. 2003.
- [21] P. Graham. A plan for spam, Aug. 2002. http://www.paulgraham.com/spam.html/.
- [22] A. Gulbransen, P. Vixie, and L. Esibov. RFC 2782: A DNS RR for specifying the location of services (DNS SRV). Internet Engineering Task Force (IETF) Standards Track, Feb. 2000.
- [23] J. Kong, P. O. Boykin, B. Rezaei, N. Sarshar, and V. Roychowdhury. Let your cyberalter ego share information and manage spam, May 2005. http://arxiv.org/abs/physics/ 0504026.
- [24] B. Laurie and R. Clayton. Proof-of-work proves not to work. The Third Annual Workshop on Economics and Information Security, May 2004
- [25] T. Loder, M. V. Alstyne, and R. Wash. An economic answer to unsolicited communication. In ACM Conference on Electronic Commerce, May 2004.
- [26] J. Lyon and M. Wong. Sender ID: Authenticating E-Mail. Internet Engineering Task Force Draft IETF, Oct. 2004.
- [27] D. Mazières. A toolkit for user-level file systems. In USENIX Technical Conference, June 2001.
- [28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Advances in Cryptology — EUROCRYPT 99, May 1999.
- [29] V. Prakash. Razor. http://razor.sourceforge.net.
- [30] S. Radicati. Anti-spam market trends, 2003-2007. Radicati Group Study, 2003. http://www.radicati.com/.
- [31] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.
- [32] M. Walfish, J. D. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control. In 3rd Symposium on Networked System Design and Implementation (NSDI), San Jose, CA, May 2006.
- [33] M. W. Wong. Sender authentication: What to do, July 2005. http://spf.pobox.com/whitepaper.pdf.
- [34] W. Yerazunis. The spam-filter accuracy plateau at 99.9% accuracy and how to get past it. In MIT Spam Conference, Jan. 2004.
- [35] P. Zimmermann. The Official PGP User's Guide. MIT Press, Cambridge, 1995.

PRESTO: Feedback-driven Data Management in Sensor Networks

Ming Li, Deepak Ganesan, and Prashant Shenoy

Department of Computer Science, University of Massachusetts, Amherst MA 01003.

{mingli,dganesan,shenoy}@cs.umass.edu

Abstract

This paper presents PRESTO, a novel two-tier sensor data management architecture comprising proxies and sensors that cooperate with one another for acquiring data and processing queries. PRESTO proxies construct time-series models of observed trends in the sensor data and transmit the parameters of the model to sensors. Sensors check sensed data with model-predicted values and transmit only deviations from the predictions back to the proxy. Such a model-driven push approach is energyefficient, while ensuring that anomalous data trends are never missed. In addition to supporting queries on current data, PRESTO also supports queries on historical data using interpolation and local archival at sensors. PRESTO can adapt model and system parameters to data and query dynamics to further extract energy savings. We have implemented PRESTO on a sensor testbed comprising Intel Stargates and Telos Motes. Our experiments show that in a temperature monitoring application, PRESTO yields one to two orders of magnitude reduction in energy requirements over on-demand, proactive or model-driven pull approaches. PRESTO also results in an order of magnitude reduction in query latency in a 1% duty-cycled five hop sensor network over a system that forwards all queries to remote sensor nodes.

1 Introduction

1.1 Motivation

Networked data-centric sensor applications have become popular in recent years. Sensors sample their surrounding physical environment and produce data that is then processed, aggregated, filtered, and queried by the application. Sensors are often untethered, necessitating efficient use of their energy resources to maximize application lifetime. Consequently, energy-efficient data management is a key problem in sensor applications.

Data management approaches in sensor networks have centered around two competing philosophies. Early efforts such as Directed Diffusion [11] and Cougar [23] espoused the notion of the sensor network as a database. The framework assumes that intelligence is placed at the sensors and that queries are pushed deep into the network, possibly all the way to the remote sensors. Direct querying of remote sensors is energy efficient, since query processing is handled at (or close to) the data source, thereby reducing communication needs. However, the approach assumes that remote sensors have sufficient processing resources to handle query processing, an assumption that may not hold in untethered networks of inexpensive sensors (e.g., Berkeley Motes [19]). In contrast, efforts such as TinyDB [13] and acquisitional query processing [3] from the database community have adopted an alternate approach. These efforts assume that intelligence is placed at the edge of the network, while keeping the sensors within the core of the network simple. In this approach, data is pulled from remote sensors by edge elements such as base-stations, which are assumed to be less resource- and energy-constrained than remote sensors. Sensors within the network are assumed to be capable of performing simple processing tasks such as in-network aggregation and filtering, while complex query processing is left to base stations (also referred to as micro-servers or sensor proxies). In acquisitional query processing [3], for instance, the base-station uses a spatio-temporal model of the data to determine when to pull new values from individual sensors; data is refreshed from remote sensors whenever the confidence intervals on the model predictions exceed query error tolerances.

While both of these philosophies inform our present work, existing approaches have several drawbacks: Need to capture unusual data trends: Sensor applications need to be alerted when unusual trends are observed in the sensor field; for instance, a sudden increase in temperature may indicate a fire or a break-down in airconditioning equipment. Although rare, it is imperative for applications, particularly those used for monitoring, to detect these unusual patterns with low latency. Both

TinyDB [13] and acquisitional query processing [3] rely on a pull-based approach to acquire data from the sensor field. A pure pull-based approach can never guarantee that all unusual patterns will be always detected, since the anomaly may be confined between two successive pulls. Further, increasing the pull frequency to increase anomaly detection probability has the harmful side-effect of increasing energy consumption at the untethered sensors.

Support for archival queries: Many existing efforts focus on querying and processing of current (live) sensor data, since this is the data of most interest to the application. However, support for querying historical data is also important in many applications such as surveillance, where the ability to retroactively "go back" is necessary, for instance, to determine how an intruder broke into a building. Similarly, archival sensor data is often useful to conduct postmortems of unusual events to better understand them for the future. Architectures and algorithms for efficiently querying archival sensor data have not received much attention in the literature.

Adaptive system design: Long-lived sensor applications need to adapt to data and query dynamics while meeting user performance requirements. As data trends evolve and change over time, the system needs to adapt accordingly to optimize sensor communication overhead. Similarly, as the workload—query characteristics and error tolerance—changes over time, the system needs to adapt by updating the parameters of the models used for data acquisition. Such adaptation is key for enhancing the longevity of the sensor application.

1.2 Research Contributions

This paper presents PRESTO, a two-tier sensor architecture that comprises sensor proxies at the higher tier, each controlling tens of remote sensors at the lower tier. PRESTO¹ proxies and sensors interact and cooperate for acquiring data and processing queries [4]. PRESTO strives to achieve energy efficiency and low query latency by exploiting resource-rich proxies, while respecting constraints at resource-poor sensors. Like TinyDB, PRESTO puts intelligence at the edge proxies while keeping the sensors inside the network simple. A key difference though is that PRESTO endows sensors with the ability to asynchronously push data to proxies rather than solely relying on pulls. Our design of PRESTO has led to the following contributions.

Model-driven Push: Central to PRESTO is the use of a feedback-based model-driven push approach to support queries in an energy-efficient, accurate and low-latency manner. PRESTO proxies construct a model that captures correlations in the data observed at each sensor. The remote sensors check the sensed data against this model and push data only when the observed data deviates from the values predicted by the model, thereby capturing anomalous trends. Such a model-driven push approach reduces communication overhead by only pushing deviations from the observed trends, while guaranteeing that unusual patterns in the data are never missed. An important requirement of our model is that it should be very inexpensive to check at resource-poor sensors, even though it can be expensive to construct at the resource-rich proxies. PRESTO employs seasonal ARIMA-based time series models to satisfy this *asymmetric* requirement.

Support for archival queries: Whereas PRESTO supports queries on current data using model-driven push, it also supports queries on historical data using a novel combination of prediction, interpolation, and local archival. By associating confidence intervals with the model predictions and caching values predicted by the model in the past, a PRESTO proxy can directly respond to such queries using cached data so long as it meets query error tolerances. Further, PRESTO employs interpolation methods to progressively refine past estimates whenever new data is fetched from the sensors. PRESTO sensors also log all observations on relatively inexpensive flash storage; the proxy can fetch data from sensor archives to handle queries whose precision requirements can not be met using the local cache. Thus, PRESTO exploits the proxy cache to handle archival queries locally whenever possible and resorts to communication with the remote sensors only when absolutely necessary.

Adaptation to Data and Query Dynamics: Long-term changes in data trends are handled by periodically refining the parameters of the model at the proxy, which improves prediction accuracy and reduces the number of pushes. Changes in query precision requirements are handled by varying the threshold used at a sensor to trigger a push. If newer queries require higher precision (accuracy), then the threshold is reduced to ensure that small deviations from the model are reported to the proxy, enabling it to respond to queries with higher precision. Overall, PRESTO proxies attempt to balance the cost of pushes and the cost of pulls for each sensor.

We have implemented PRESTO using a Stargate proxy and Telos Mote sensors. We demonstrate the benefits of PRESTO using an extensive experimental evaluation. Our results show that PRESTO can scale up to one hundred Motes per proxy. When used in a temperature monitoring application, PRESTO imposes an energy requirements that is one to two orders of magnitude less than existing techniques that advocate on-demand, proactive, or model-driven pulls. At the same time, the average latency for queries is within six seconds for a 1% duty-cycled five hop sensor network, which is an order of

¹PRESTO is an acronym for PREdictive STOrage.

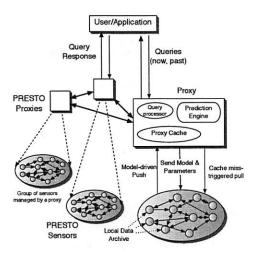


Figure 1: The PRESTO data management architecture.

magnitude less than a system that forwards all queries to remote sensor nodes, while not significantly more than a system where all queries are answered at the proxy.

The rest of this paper is structured as follows. Section 2 provides an overview of PRESTO. Sections 3 and 4 describe the design of the PRESTO proxy and sensors, respectively, while Section 5 presents the adaptation mechanisms in PRESTO. Sections 6 and 7 present our implementation and our experimental evaluation. Finally, Sections 8 and 9 discuss related work and our conclusions.

2 System Architecture

System Model: PRESTO envisions a two-tier data management architecture comprising a number of sensor proxies, each controlling several tens of remote sensors (see Figure 1). Proxies at the upper tier are assumed to be rich in computational, communication, and storage resources and can use them continuously. The task of this tier is to gather data from the lower tier and answer queries posed by users or the application. A typical proxy configuration may be comprised of an Intel Stargate [21] node with multiple radios—an 802.11 radio that connects it to an IP network and a low-power 802.15.4 radio that connects it to sensors in the lower tier. Proxies are assumed to be tethered or powered by a solar cell. A typical deployment will consist of multiple geographically distributed proxies, each managing tens of sensors in its vicinity. In contrast, PRESTO sensors are assumed to be low-power nodes, such as Telos Motes [18], equipped with one or more sensors, a microcontroller, flash storage and a wireless radio. The task of this tier is to sense data, transmit it to proxies when appropriate, while archiving all data locally in flash storage. The primary constraint at this tier is energy—sensor nodes are assumed to be untethered, and hence battery-powered, with a limited lifetime. Sensors are assumed to be deployed in a multi-hop configuration and are aggressively duty-cycled; standard multi-hop routing and duty-cycled MAC protocols can be used for this purpose. Since communication is generally more expensive than processing or storage [5], PRESTO sensors attempt to trade communication for computation or storage, whenever possible.

System Operation: Assuming such an environment, each PRESTO proxy constructs a model of the data observed at each sensor. The model uses correlations in the past observations to predict the value likely to be seen at any future instant t. The model and its parameters are transmitted to each sensor. The sensor then executes the model as follows: at each sampling instant t, the actual sensed value is compared to the value predicted by the model. If the difference between the two exceed a threshold, the model is deemed to have "failed" to accurately predict that value and the sensed value is pushed to the proxy. In contrast, if the difference between the two is smaller than a threshold, then the model is assumed to be accurate for that time instant. In this case, the sensor archives the data locally in flash storage and does not transmit it to the proxy. Since the model is also known to the proxy, the proxy can compute the predicted value and use it as an approximation of the actual observation when answering queries. Thus, so long as the model accurately predicts observed values, no communication is necessary between the sensor and the proxy; the proxy continues to use the predictions to respond to queries. Further, any deviations from the model are always reported to the proxy and anomalous trends are quickly detected as a result.

Given such a model-driven push technique, a query arriving at the proxy is processed as follows. PRESTO assumes that each query specifies a tolerance on the error it is willing to accept. Our models are capable of generating a confidence interval for each predicted value. The PRESTO proxy compares the query error tolerance with the confidence intervals and uses the model predictions so long at the query error tolerance is not violated. If the query demands a higher precision, the proxy simply pulls the actual sensed values from the remote sensors and uses these values to process the query. Every prediction made by the model is cached at the proxy; the cache also contains all values that were either pushed or pulled from the remote sensors. This cached data is used to respond to historical queries so long as query precision is not violated, otherwise the corresponding data is pulled from the local archive at the sensors.

Since trends in sensed values may change over time, a model constructed using historical data may no longer reflect current trends. A novel aspect of PRESTO is that it updates the model parameters online so that the model

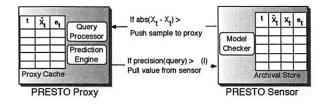


Figure 2: The PRESTO proxy comprises a prediction engine, query processor and a cache of predicted and real sensor values. The PRESTO sensor comprises a model checker and an archive of past samples with the model predictions.

can continue to reflect current observed trends. Upon receiving a certain number of updates from a sensor, the proxy uses these new values to refine the parameters of the model. These parameters are then conveyed back to the corresponding sensor, when then uses them to push subsequent values. Thus, our approach incorporates active feedback between the proxy and each sensor—the model parameters are used to determine which data values get pushed to the proxy, and the pushed values are used to compute the new parameters of the model. If the precision demanded by queries also changes over time, the thresholds used by sensors to determine which values should be pushed are also adapted accordingly—higher precision results in smaller thresholds. Next, we present the design of the PRESTO proxy and sensor in detail.

3 PRESTO Proxy

The PRESTO proxy consists of four key components (see Figure 2): (i) modeling and prediction engine, which is responsible for determining the initial model parameters, periodic refinement of model parameters, and prediction of data values likely to be seen at the various sensors, (ii) query processor, which handles queries on both current and historical data, (iii) local cache, which is a cache of all data pushed or pulled by sensors as well as all past values predicted by the model, and (iv) a fault detector, which detects sensor failures. We describe each component in detail in this section.

3.1 Modeling and Prediction Engine

The goal of the modeling and prediction engine is to determine a model, using a set of past sensor observations, to forecast future values. The key premise is that the physical phenomena observed by sensors exhibit long-term and short-term correlations and past values can be used to predict the future. This is true for weather phenomena such as temperature that exhibit long-term seasonal variations as well as short-term time-of-day and hourly variations. Similarly phenomena such as traf-

fic at an intersection exhibit correlations based on the hour of the day (e.g., traffic peaks during "rush" hours) and day of the week (e.g., there is less traffic on weekends). PRESTO proxies rely on seasonal ARIMA models; ARIMA is a popular family of time-series models that are commonly used for studying weather and stock market data. Seasonal ARIMA models (also known as SARIMA) are a class of ARIMA models that are suitable for data exhibiting seasonal trends and are well-suited for sensor data. Further they offer a way to deal with nonstationary data i.e. whose statistical properties change over time [1]. Last, as we demonstrate later, while seasonal ARIMA models are computationally expensive to construct, they are inexpensive to check at the remote sensors—an important property we seek from our system. The rest of this section presents the details of our SARIMA model and its use within PRESTO.

Prediction Model: A discrete time series can be represented by a set of time-ordered data $(x_{t_1}, x_{t_2}, ..., x_{t_n})$, resulting from observation of some temporal physical phenomenon such as temperature or humidity. Samples are assumed to be taken at discrete time instants $t_1, t_2, ...$ The goal of time-series analysis is to obtain the parameters of the underlying physical process that governs the observed time-series and use this model to forecast future values.

PRESTO models the time series of observations at a sensor as an Autoregressive Integrated Moving Average (ARIMA) process. In particular, the data is assumed to conform to the Box-Jenkins SARIMA model [1]. While a detailed discussion of SARIMA models is outside the scope of this paper, we provide the intuition behind these models for the benefit of the reader. An SARIMA process has four components: auto-regressive (AR), movingaverage (MA), one-step differencing, and seasonal differencing. The AR component estimates the current sample as a linear weighted sum of previous samples; the MA component captures relationship between prediction errors; the one-step differencing component captures relationship between adjacent samples; and the seasonal differencing component captures the diurnal, monthly, or yearly patterns in the data. In SARIMA, the MA component is modeled as a zero-mean, uncorrelated Gaussian random variable (also referred to as white noise). The AR component captures the temporal correlation in the time series by modeling a future value as a function of a number of past values.

In its most general form, the Box-Jenkins seasonal model is said to have an order $(p,d,q) \times (P,D,Q)_S$; the order of the model captures the dependence of the predicted value on prior values. In SARIMA, p and q are the orders of the auto-regressive (AR) and moving average (MA) processes, P and Q are orders of the seasonal AR and MA components, d is the order of differencing, D is

the order of seasonal differencing, and S is the seasonal period of the series. Thus, SARIMA is family of models depending on the integral values of p, q, P, Q, d, D, S. ²

Model Identification and Parameter Estimation: Given the general SARIMA model, the proxy needs to determine the order of the model, including the order of differential and the order of auto-regression and moving average. That is, the values of p, d, q, P, D and Q need to be determined. This step is called model identification and is typically performed once during system initialization. Model identification is well documented in most time series textbooks [1] and we only provide a high level overview here. Intuitively, since the general model is actually a family of models, depending on the values of p, q, etc., this phase identifies a particular model from the family that best captures the variations exhibited by the underlying data. It is somewhat analogous to fitting a curve on a set of data values. Model identification involves collecting a sample time series from the field and computing its auto-correlation function (ACF) and partial auto-correlation function (PACF). A series of tests are then performed on the ACF and the PACF to determine the order of the model [1].

Our analysis of temperature traces has shown that the best model for temperature data is a Seasonal ARIMA of order $(0,1,1) \times (0,1,1)_S$. The general model in Equation 1 reduces to

$$(1 - B)(1 - B^S)X_t = (1 - \theta B)(1 - \Theta B^S)e_t$$
 (2)

where θ and Θ are parameters of this $(0,1,1)\times(0,1,1)_S$ SARIMA model and capture the variations shown by different temperature traces. B is the backward operator and is short-hand for $B^iX_t=X_{t-i}$. S is the seasonal period of the time series and e_t is the prediction error.

When employed for a temperature monitoring application, PRESTO proxies are seeded with a $(0,1,1) \times (0,1,1)_S$ SARIMA model. The seasonal period S is also seeded. The parameters θ and Θ are then computed by the proxy during the initial training phase before the system becomes operational. The training phase involves gathering a data set from each sensor and using the least squares method to estimate the values of parameters θ and Θ on a per-sensor basis (see [1] for the detailed procedure for estimating these parameters). The order of the model and the values of θ and Θ are then conveyed to each sensor. Section 5 explains how θ and Θ can be periodically refined to adapt to any long-term changes in the

$$\Phi_P(B^S) \cdot \phi_P(B) \cdot (1-B)^d (1-B^S)^D X_t = \theta_q(B) \Theta_Q(B^S) e_t$$
 (1)

where B is the backward operator such that $B^iX_t = X_{t-i}$, S is the seasonal period, θ , Θ are parameters of the model, and e_t is the prediction error.

sensed data that occurs after the initial training phase.

Model-based Predictions: Once the model order and its parameters have been determined, using it for predicting future values is a simple task. The predicted value X_t for time t is simply given as:

$$X_{t} = X_{t-1} + X_{t-S} - X_{t-S-1} + \theta e_{t-1} - \Theta e_{t-S} + \theta \Theta e_{t-S-1}$$
(3)

where θ and Θ are known parameters of the model, X_{t-1} denotes the previous observation, X_{t-S} and X_{t-S-1} denotes the values seen at this time instant and the previous time instant in the previous season. For temperature monitoring, we use a seasonal period S of one day, and hence, X_{t-S} and X_{t-S-1} represent the values seen yesterday at this time instant and the previous time instant, respectively. e_{t-k} denotes the prediction error at time t-k (the prediction error is simply the difference between the predicted and observed value for that instant).

Since PRESTO sensors push a value to the proxy only when it deviates from the prediction by more than a threshold, the actual values of X_{t-1} , X_{t-S} and X_{t-S-1} seen at the sensor may not be known to the proxy. However, since the lack of a push indicates that the model predictions are accurate, the proxy can simply use the corresponding model predictions as an approximation for the actual values in Equation 3. In this case, the corresponding prediction error e_{t-k} is set to zero. In the event X_{t-1} , X_{t-S} or X_{t-S-1} were either pushed by the sensor or pulled by the proxy, the actual values and the actual prediction errors can used in Equation 3.

Both the proxy and the sensors use Equation 3 to predict each sampled value. At the proxy, the predictions serve as a substitute for the actual values seen by the sensor and are used to answer queries that might request the data. At the sensor, the prediction is used to determine whether to push—the sensed value is pushed only if the prediction error exceeds a threshold δ .

Finally, we note the *asymmetric* property of our model. The initial model identification and parameter estimation is a compute-intensive task performed by the proxy. Once determined, predicting a value using the model *involves no more than eight floating point operations* (three multiplications and five additions/subtractions, as shown in Equation 3). This is inexpensive even on resource-poor sensor nodes such as Motes and can be approximated using fixed point arithmetic.

3.2 Query Processing at a Proxy

In addition to forecasting future values, the prediction engine at the proxy also provides a confidence interval for each predicted value. The confidence interval represents a bound on the error in the predicted value and is crucial for query processing at the proxy. Since each

 $^{^2}$ While not essential for our discussion, we present the general Box-Jenkins seasonal model for sake of completeness. The general model of order $(p,d,q) \times (P,D,Q)_S$ is given by the equation

query arrives with an error tolerance, the proxy compares the error tolerance of a query with the confidence interval of the predictions, and the current push threshold, δ . If the confidence interval is tighter than the error tolerance, then the predicted values are sufficiently accurate to respond to the query. Otherwise the actual value is fetched from the remote sensor to answer the query. Thus, many queries can be processed locally even if the requested data was never reported by the sensor. As a result, PRESTO can ensure low latencies for such queries without compromising their error tolerance. The processing of queries in this fashion is similar to that proposed in the BBQ data acquisition system [3], although there are significant differences in the techniques.

For a Seasonal ARIMA $(0,1,1) \times (0,1,1)_S$ model, the confidence interval of l step ahead forecast, $\lambda(l)$ is:

$$\lambda(l) = \pm u_{\varepsilon/2} (1 + \sum_{i=1}^{l-1} (1 - \theta)^2)^{1/2} \sigma \tag{4}$$

where $u_{\varepsilon/2}$ is value of the unit Normal distribution at $\varepsilon/2$, σ is the variance of 1 step ahead prediction error.

3.3 Proxy Cache

Each proxy maintains a cache of previously fetched or predicted data values for each sensor. Since storage is plentiful at the proxy—microdrives or hard-drives can be used to hold the cache—the cache is assumed to be infinite and all previously predicted or fetched values are assumed to be stored at the proxy. The cache is used to handle queries on historical data—if requested values have already been fetched or if the error bounds of cached predictions are smaller than the query error tolerance, then the query can be handled locally, otherwise the requested data is pulled from the archive at the sensor. After responding to the query, the newly fetched values are inserted into the cache for future use.

A newly fetched value, upon insertion, is also used to improve the accuracy of the neighboring predictions using interpolation. The intuition for using interpolation is as follows. Upon receiving a new value from the sensor, suppose that the proxy finds a certain prediction error. Then it is very likely that the predictions immediately preceding and following that value incurred a similar error, and interpolation can be used to scale those cached values by the prediction error, thereby improving their estimates. PRESTO proxies currently use two types of interpolation heuristics: forward and backward.

Forward interpolation is simple. The proxy uses Equation 3 to predict the values and Equation 4 to re-estimate the confidence intervals for all samples between the newly inserted value and the next pulled or pushed value. In backward interpolation, the proxy scans backwards from the newly inserted value and modifies all cached

predictions between the newly inserted value and the previous pushed or pulled value. To do so, it makes a simplifying assumption that the prediction error grows linearly at each step, and the corresponding prediction error is subtracted from each prediction.

$$X_{t}' = X_{t} - \frac{t - T'}{T - T'}e_{T} \tag{5}$$

where X_t is the original prediction, X_t' is the updated prediction, T denotes the observation instant of the newly inserted value, T' is time of the nearest pushed or pulled value before T.

3.4 Failure Detection

Sensors are notoriously unreliable and can fail due hard-ware/software glitches, harsh deployment conditions or battery depletion. Our predictive techniques limit message exchange between a proxy and a sensor, thereby reducing communication overhead. However, reducing message frequency also affects the latency to detect sensor failures and to recover from them. In this work, we discuss mechanisms used by the PRESTO proxy to detect sensor failures. Failure recovery can use techniques such as spatial interpolation, which are outside the scope of this paper.

The PRESTO proxy flags a failure if pulls or feedback messages are not acknowledged by a sensor. This use of implicit heartbeats has low communication energy overhead, but provides an interesting benefit. A pull is initiated by the proxy depending on the confidence bounds, which in turn depends on the variability observed in the sensor data. Consequently, failure detection latency will be lower for sensors that exhibit higher data variability (resulting in more pushes or pulls). For sensors that are queried infrequently or exhibit low data variability, the proxy relies on the less-frequent model feedback messages for implicit heartbeats; the lack of an acknowledgment signals a failure. Thus, proxy-initiated control or pull messages can be exploited for failure detection at no additional cost; the failure detection latency depends on the observed variability and confidence requirements of incoming queries. Explicit heartbeats can be employed for applications with more stringent needs.

4 PRESTO Sensor

PRESTO sensors perform three tasks: (i) use the model predictions to determine which observations to push, (ii) maintain a local archive of all observations, and (iii) respond to pull requests from the proxy.

The PRESTO sensor acts as a mirror for the prediction model at the proxy—both the proxy and the sensor execute the model in a completely identical fashion. Consequently, at each sampling instant, the sensor knows the exact estimate of the sampled value at the proxy and can determine whether the estimate is accurate. Only those samples that deviate significantly from the prediction are pushed. As explained earlier, the proxy transmits all the parameters of the model to each sensor during system initialization. In addition, the proxy also specifies a threshold δ that defines the worst-case deviation in the model prediction that the proxy can tolerate. Let X_t denote the actual observation at time t and let \hat{X}_t denote the predicted value computed using Equation 3. Then,

If
$$|\hat{X}_t - X_t| > \delta$$
, Push X_t to Proxy. (6)

As indicated earlier, computation of \hat{X}_t using Equation 3 involves reading of a few past values such as X_{t-S} from the archive in flash storage and a few floating point multiplications and additions, all of which are inexpensive.

PRESTO sensors archive all sensed values into an energy-efficient NAND flash store; the flash archive is a log of tuples of the form: (t, X_t, \hat{X}_t, e_t) . A simple index is maintained to permit random access to any entry in the log. A pull request from a proxy involves the use of this index to locate the requested data in the archive, followed by a read and a transmit.

5 Adaptation in PRESTO

PRESTO is designed to adapt to long-term changes in data and query dynamics that occur in any long-lived sensor application. To enable system operation at the most energy-efficient point, PRESTO employs active feedback from proxies to sensors; this feedback takes two forms—adaptation to data and query dynamics.

5.1 Adaptation to Data Dynamics

Since trends in sensor observation may change over time, a model constructed using historical data may no longer reflect current trends—the model parameters become stale and need to be updated to regain energy-efficiency. PRESTO proxies periodically retrain the model in order to refine its parameters. The retraining phase is similar to the initial training—all data since the previous retraining phase is gathered and the least squares method is used to recompute the model parameters θ and Θ [1]. The key difference between the initial training and the retraining lies in the data set used to compute model parameters.

For the initial training, an actual time series of sensor observations is used to compute model parameters. However, once the system is operational, sensors only report observations when they significantly deviate from the predicted values. Consequently, the proxy only has access to a small subset of the observations made at each sensor. Thus, the model must be retrained with *incomplete information*. The time series used during the retraining phase contains all values that were either pushed

or pulled from a sensor; all missing values in the time series are substituted by the corresponding model predictions. Note that these prior predictions are readily available in the proxy cache; furthermore, they are guaranteed to be a good approximation of the actual observations (since these are precisely the values for which the sensor did not push the actual observations). This approximate time series is used to retrain the model and recompute the new parameters.

For the temperature monitoring application that we implemented, the models are retrained at the end of each day.³ The new parameters θ and Θ are then pushed to each sensor for future predictions. In practice, the parameters need to be pushed only if they deviate from the previously computed parameters by a non-trivial amount (i.e., only if the model has actually changed).

5.2 Adaptation to Query Dynamics

Just as sensor data exhibits time-varying behavior, query patterns can also change over time. In particular, the query tolerance demanded by queries may change over time, resulting in more or fewer data pulls. The proxy can adapt the value of the threshold parameter δ in Equation 6 to directly influence the fraction of queries that trigger data pulls from remote sensors. If the threshold δ is large relative to the mean error tolerance of queries, then the number of pushes from the sensor is small and the number of pulls triggered by queries is larger. If δ is small relative to the query error tolerance, then there will be many wasteful pushes and fewer pulls (since the cached data is more precise than is necessary to answer the majority of queries). A careful selection of the threshold parameter δ allows a proxy to balance the number of pushes and the number of pulls for each sensor.

To handle such query dynamics, the PRESTO proxy uses a moving window average to track the mean error tolerance of queries posed on the sensor data. If the error tolerance changes by more than a pre-defined threshold, the proxy computes a new δ and transmits it to the sensor so that it can adapt to the new query pattern.

6 PRESTO Implementation

We have implemented a prototype of PRESTO on a multi-tier sensor network testbed. The proxy tier employs Crossbow Stargate nodes with a 400MHz Intel XScale processor and 64MB RAM. The Stargate runs the Linux 2.4.19 kernel and EmStar release 2.1 and is equipped with two wireless radios, a Cisco Aironet 340-based 802.11b radio and a hostmote bridge to the Telos mote sensor nodes using the EmStar transceiver. The

³Since the seasonal period is set to one day, this amounts to a retraining after each season.

sensor tier uses Telos Mote sensor nodes, each consisting of a MSP430 processor, a 2.4GHz CC2420 radio, and 1MB external flash memory. The sensor nodes run TinyOS 1.1.14. Since sensor nodes may be several hops away from the nearest proxy, the sensor tier employs MultiHopLEPSM multi-hop routing protocol from the TinyOS distribution to communicate with the proxy tier.

Sensor Implementation: Our PRESTO implementation on the Telos Mote involves three major tasks: (i) model checking, (ii) flash archival, and (ii) data pull. A simple data gathering task periodically obtains sensor readings and sends the sample to the model checker. The model checking task uses the most recent model parameters (θ and Θ) and push delta (δ) obtained from the proxy to determine if a sample should be pushed to the proxy as per Equation 6. Each push message to the proxy contains the id of the mote, the sampled data, and a timestamp recording the time of the sampling. Upon a pull from the proxy, the model checking task performs the forward and backward updates to ensure consistency between the proxy and sensor view. For each sample, the archival task stores a record to the local flash that has three fields: (i) the timestamp when the data was sampled, (ii) the sample itself, and (iii) the predicted value from the model checker. The final component of our sensor implementation is a pull task that, upon receiving a pull request, reads the corresponding data from the flash using a temporal index-based search, and responds to the proxy.

Proxy Implementation: At the core of the proxy implementation is the prediction engine. The prediction engine includes a full implementation of ARIMA parameter estimation, prediction and update. The engine uses two components, a cache of real and predicted samples, and a protocol suite that enables interactions with each sensor. The proxy cache is a time-series stream of records, each of which includes a timestamp, the predicted sensor value, and the prediction error. The proxy uses one stream per node that it is responsible for, and models each node's data separately. The prediction engine communicates with each sensor using a protocol suite that enables it to provide feedback and change the operating parameters at each sensor.

Queries on our system are assumed to be posed at the appropriate proxy using either indexing [5] or routing [12] techniques. A query processing task at the proxy accepts queries from users, checks whether it can be answered by the prediction engine based on the local cache. If not, a pull message is sent to the corresponding sensor.

Our proxy implementation includes two enhancements to the hostmote transceiver that comes with the EmStar distribution [6]. First, we implemented a priority-based 64-length FIFO outgoing message queue in the transceiver to buffer pull requests to the sensors. There are two priority levels — the higher priority cor-

responds to parameter feedback messages to the sensor nodes, and the lower priority corresponds to data pull messages. Prioritizing messages ensures that parameter messages are not dropped even if the queue is full as a result of excess pulls. Our second enhancement involves emulating the latency characteristics of a dutycycling MAC layer. Many MAC-layer protocols have been proposed for sensor networks such as BMAC [17] and SMAC [24]. However, not all these MAC layers are supported on all platforms — for instance, neither BMAC nor SMAC is currently supported on the Telos Motes that we use. We address this issue by benchmarking the latency introduced by BMAC on Mica2 sensor nodes, and using these measurements to drive our experiments. Thus, the proxy implementation includes a MAC-layer emulator that adds duty-cycling latency corresponding to the chosen MAC duty-cycling parameters.

7 Experimental Evaluation

In this section, we evaluate the performance of PRESTO using our prototype and simulations. The testbed for our experiments comprises one Stargate proxy and twenty Telos Mote sensor nodes. One of the Telos motes is connected to a Stargate node running a sensor network emulator in Emstar[8]. This emulator enables us to introduce additional virtual sensor nodes in our large-scale experiments that share a single Telos mote radio as the transceiver to send and receive messages. In addition to the testbed, we use numerical simulations in Matlab to evaluate the performance of the data processing algorithms in PRESTO.

Our experiments involve both replays of previously gathered sensor data as well as a live deployment. The first set of experiments are trace-driven and use a seven day temperature dataset from James reserve [22]. The first two days of this trace are used to train the model. In our experiments, sensors use the values from the remainder of these traces—which are stored in flash memory—as a substitute for live data gathering. This setup ensures repeatable experiments and comparison of results across experiments (which were conducted over a period of several weeks). We also experiment with a live, four day outdoor deployment of PRESTO at UMass to demonstrate that our results are representative of the "real world".

In order to evaluate the query processing performance of PRESTO, we generate queries as a Poisson arrival process. Each query requests the value of the temperature at a particular time that is picked in a uniform random manner from the start of the experiment to the current time. The confidence interval requested by the query is chosen from a normal distribution.

7.1 Microbenchmarks

Our first experiment involves a series of microbenchmarks of the energy consumption of communication, processing and storage to evaluate individual components of the PRESTO proxy and sensors. These microbenchmarks are based on measurements of two sensor platforms — a Telos mote, and a Mica2 mote augmented with a NAND flash storage board fabricated at UMass. The board is attached to the Mica2 mote through the standard 51-pin connector, and provides a considerably more energy-efficient storage option than the AT45DB041B NOR flash that is loaded by default on the Mica2 mote [15]. The NAND flash board enables the PRESTO sensor to archive a large amount of historical data at extremely low energy cost.

Module	Component	Operation	Energy
NAND flash-	NAND Flash	Read + Write + Erase 1 sample	21nJ
enabled Mica2	ATmega128L Processor	1 Prediction	240nJ
	CC1000 Radio	Transmit 1 sample + Receive 1 ACK	$20.3 \mu J$
Telos Mote	ST M25P80 Flash	Read + Write + Erase 1 sample	2.14μJ
	MSP430 Processor	1 Prediction	27nJ
	CC2420 Radio	Transmit 1 sample + Receive 1 ACK	3.3μ J

Table 1: Energy micro-benchmarks for sensor nodes.

	Round Trip Latency(ms)		
Routing Hops	1%	7.53%	35.5%
1-hop	2252	350	119
2-hop	4501	695	235
3-hop	6750	1040	347
4-hop	8999	1388	465
5-hop	11249	1733	580

Table 2: Round trip latencies using B-MAC

Energy Consumption: We measure the energy consumption of three components—computation per sample at the sensor, communication for a push or pull, and storage for reads, writes and erases. Table 1 shows that the results depend significantly on the choice of platform. On the Mica2 mote with external NAND flash, storage of a sample in flash is an order of magnitude more efficient than the ARIMA prediction computation, and three orders of magnitude more efficient than communicating a sample over the CC1000 radio. The Telos mote uses a more energy-efficient radio (CC2420) and processor (TI MSP 430), but a less efficient flash than the modified Mica2 mote. On the Telos mote, the prediction computation is the most energy-efficient operation, and is 80 times more efficient than storage, and 122 times more efficient than communication. The high cost of storage on the Telos mote makes it a bad fit for a storage-centric architecture such as PRESTO.

In order to fully exploit state-of-art in computation, communication and storage, a new platform is required that combines the best features of the two platforms that we have measured. This platform would use the TI MSP 430 microcontroller and CC2420 radio on the Telos mote together with NAND flash storage. Assuming that the component-level microbenchmarks in Table 1 hold for the new platform, storage and computation would be roughly equal cost, whereas communication would be two to three orders of magnitude more expensive than both storage and communication. We note that the energy requirements for communication in all the above benchmarks would be even greater if one were to include the overhead due to duty-cycling, packet headers and multi-hop routing. These comparisons validate our key premise that in future platforms, storage will offer a more energy-efficient option than communication and should be exploited to achieve energy-efficiency.

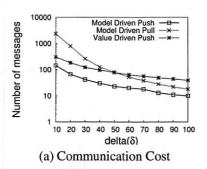
Component		Operation	Latency	Energy
Stargate (PX	A255)	Model Estimation	21.75ms	11mJ
Telos (MSP430)	Mote	Predict One Sample	18μs	27nJ

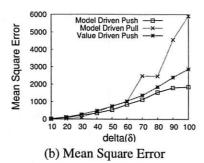
Table 3: Asymmetry: Model estimation vs Model checking

Communication Latency: Our second microbenchmark evaluates the latency of directly querying a sensor node. Sensor nodes are often highly duty-cycled to save energy, i.e. their radios are turned off to reduce energy use. However, as shown in Table 2, better dutycycling corresponds to increased duration between successive wakeups and worse latency for the CC1000 radio on the Mica2 node. For typical sensor network dutycycles of 1% or less, the latency is of the order of many seconds even under ideal 100% packet delivery conditions. Under greater packet-loss rates that are typical of wireless sensor networks [25], this latency would increase even further. We are unable to provide numbers for the CC2420 radio on the Telos mote since there is no available TinyOS implementation of an energy-efficient MAC layer with duty-cycling support for this radio.

Our measurements validate our claim that directly querying a sensor network incurs high latency, and this approach may be unsuitable for interactive querying. To reduce querying latency, the proxy should handle as many of the queries as possible.

Asymmetric Resource Usage: Table 3 demonstrates how PRESTO exploits computational resources at the proxy and the sensor. Determining the parameters of the ARIMA model at the proxy is feasible for a Stargate-class device, and requires only 21.75 ms per sensor. This operation would be very expensive, if not infeasible, on





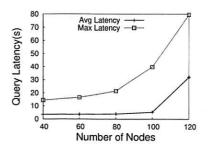


Figure 3: Comparison of PRESTO SARIMA models with model-driven pull and value-driven push.

Figure 4: Scalability of PRESTO: Impact of network size.

a Telos Mote due to resource limitations. In contrast, checking if the model is correct at the Mote consumes considerably less energy since it consists of only three floating point multiplications (approximated using fixed point arithmetic) and five additions/subtractions corresponding to Equation 3. This validates the design choice in PRESTO to separate model-building from model-checking and to exploit proxy resources for the former and resources at the sensor for the latter.

Summary: Our microbenchmarks validate three design choices made by PRESTO—the need for a storage-centric architecture that exploits energy-efficient NAND flash storage, the need for proxy-centric querying to deal with high latency of duty-cycled radios, and exploiting proxy resources to construct models while performing only simple model-checking at the sensors.

7.2 Performance of Model-Driven Push

In this section, we validate our claim that intelligently exploiting both proxy and sensor resources offers greater energy benefit than placing intelligence only at the proxy or only at the sensor. We compare the performance of model-driven push used in PRESTO against two other data-acquisition algorithms. The first algorithm, modeldriven pull, is representative of the class of techniques where intelligence is placed solely at the proxy. This algorithm is motivated by the approach proposed in BBQ [3]. In this algorithm, the proxy uses a model of sensor data to predict future data and estimate the confidence interval in the prediction. If the confidence interval exceeds a pre-defined threshold (δ), the proxy will pull data from the sensor nodes, thus keeping the confidence interval bounded. The sensor node is simple in this case, and performs neither local storage nor model processing. While BBQ uses multi-variate Gaussians and dynamic Kalman Filters in their model-driven pull, our model-driven pull uses ARIMA predictions to ensure that the results capture the essential difference between the techniques and not the difference between the models used. The second algorithm that we compare against is a relatively naive

value-driven push. Here, the sensor node pushes the data to the proxy when the difference between current data and last pushed data is larger than a threshold (δ) . The proxy assumes that the sensor value does not change until the next push from the sensor. In general, a pull requires two messages, a request from the proxy to the sensor and a response, whereas push requires only a single message from the sensor to the proxy.

We compared the three techniques using Matlab simulations that use real data traces from James Reserve. Each experiment uses 5 days worth of data and each data point is the average of 10 runs. Figure 3 compares these three techniques in terms of the number of messages transmitted and mean-square error of predictions. In communication cost, PRESTO out-performs both the other schemes irrespective of the choice of δ . When δ is 100, the communication cost of PRESTO is half that of model-driven pull, and 25% that of value-driven push. At the same time, the mean square error in PRESTO is 30% that of model-driven pull, and 60% that of value driven push. As δ decreases, the communication cost increases for all three algorithms. However, the increase in communication cost for model-driven pull is higher than that for the other two algorithms. When δ is 50, value driven push begins to out perform model-driven pull. When δ reaches 10, the number of messages in model-driven pull is 20 times more than that of PRESTO, and 8 times more than that of value driven push. This is because in the case of model-driven pull, the proxy pulls samples from the sensor whenever the prediction error exceeds δ . However, since the prediction error is often an overestimate and since each pull is twice as expensive as a push, this results in a larger number of pull messages compared to PRESTO and value-driven push. The accuracies of the three algorithms become close to each other when δ decreases. When δ is smaller than 40, model-driven pull has slightly lower mean square error than PRESTO but incurs 4 times the number of messages.

Summary: These performance numbers demonstrate that model-driven push combines the benefits of both

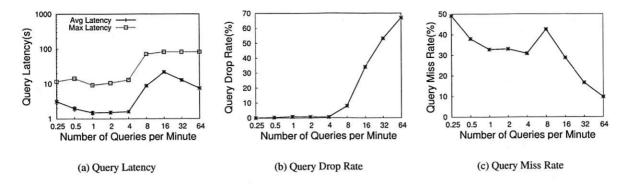


Figure 5: Scalability of PRESTO: Impact of query rates.

proxy-centric as well as sensor-centric approaches. It is 2-20 times more energy-efficient and upto 3 times more accurate than proxy-centric model-driven pull. In addition, PRESTO is upto 4 times more energy-efficient than a sensor-centric value-driven push approach.

7.3 PRESTO Scalability

Scalability is an important criteria for sensor algorithm design. In this section, we evaluate scalability along two axes — network size and the number of queries posed on a sensor network. Network size can vary depending on the application (e.g: the Extreme Scaling deployment [7] used 10,000 nodes, whereas the Great Duck Island deployment [14] used 100 nodes). The querying rate depends on the popularity of sensor data, for instance, during an event such as an earthquake, seismic sensors might be heavily queried while under normal circumstances, the query load can be expected to be light.

The testbed used in the scalability experiments comprises one Stargate proxy, twenty Telos mote sensor nodes, and an EmStar emulator that enables us to introduce additional virtual sensor nodes and perform larger scale experiments. Messages are exchanged between each sensor and the proxy through a multihop routing tree rooted at the proxy. Each sensor node is assumed to be operating at 1% duty-cycling. Since MAC layers that have been developed for the Telos mote do not currently support duty-cycling, we emulate a duty-cycling enabled MAC-layer. This emulator adds appropriate duty-cycling latency to each packet based on the microbenchmarks that we presented in Table 2.

7.3.1 Impact of Network Size

A good data management architecture should achieve energy-efficiency and low-latency performance even in large scale networks. Our first set of scalability experiments test PRESTO at different system scales on five days of data collected from the James Reserve deployment. Queries arrive at the proxy as as a Poisson process at the rate of one query/minute per sensor. The confidence interval of queries is chosen from a normal distribution, whose expectation is equal to the push threshold, $\delta=100$.

Figure 4 shows the query latency and query drop rate at system sizes ranging from 40 to 120. For system sizes of less than 100, the average latency is always below five seconds and has little variation. When the system size reaches 120, the average latency increases five-fold to 30 seconds. This is because the radio transceiver on the proxy gets congested and the queue overflows.

The effect of duty-cycling on latency is seen in Figure 4, which shows that the maximum latency increases with system scale. The maximum latency corresponds to the worst case of PRESTO when a sequence of query misses occur and result in pulls from sensors. This results in queuing of queries at the proxy, and hence greater latency. An in-network querying mechanism such as Directed Diffusion [11] that forwards every query into the network would incur even greater latency than the worst case in PRESTO since every query would result in a pull. These experiments demonstrate the benefits of modeldriven pushes for user queries. By the use of caching and models, PRESTO results in low average-case latency by providing quick responses at the proxy for a majority of queries. We note that the use of a tiered architecture makes it easy to expand system scale to many hundreds of nodes by adding more PRESTO proxies.

7.3.2 Impact of Query Rate

Our second scalability experiment stresses the query handling ability of PRESTO. We test PRESTO in a network comprising one Stargate proxy and twenty Telos mote sensor nodes under different query rates ranging from one query every four minutes to 64 queries/minute for each sensor. Each experiment is averaged over one

hour. We measure scalability using three metrics: the query latency, query miss rate, and query drop rate. A query miss corresponds to the case when it cannot be answered at the proxy and results in a pull, and a query drop results from an overflow at the proxy queue.

Figure 5 shows the result of the interplay between model accuracy, network congestion, and queuing at the proxy. To better understand this interplay, we analyze the graphs in three parts, *i.e.*, 0.25-4 queries/minute, 4-16 queries/minute and beyond 16 queries/minute.

Region 1: Between 0.25 and 4 queries/minute, the query rate is low, and neither queuing at the proxy nor network congestion is a bottleneck. As the query rate increases, greater number of queries are posed on the system and result in a few more pulls from the sensors. As a consequence, the accuracy of the model at the proxy improves to the point where it is able to answer most queries. This results in a reduction in the average latency. This behavior is also reflected in Figure 5(c), where the query miss rate reduces as the rate of queries grows.

Region 2: Between 4 and 16 queries/minute, the query rate is higher than the rate at which queries can be transmitted into the network. The queue at the proxy starts building, thereby increasing latency for query responses. This results in a sharp increase in average latency and maximum latency, as shown in Figure 5(a). This increase is also accompanied by an increase in query drop rate beyond eight queries/minute, as more queries are dropped due to queue overflow. We estimate that eight queries/minute is the breakdown threshold for our system for the parameters chosen.

Region 3: Beyond sixteen queries/minute, the system drops a significant fraction of queries due to queue overflow as shown in Figure 5(b). Strangely, for the queries that do not get dropped, both the average latency (Figure 5(a)), and the query miss rate (Figure 5(c)) drop! This is because with each pull, the model precision improves and it is able to answer a greater fraction of the queries accurately.

The performance of PRESTO under high query rate demonstrates one of its key benefits — the ability to use the model to alleviate network congestion and queuing delays. This feature is particularly important since sensor networks can only sustain a much lower query rate than tethered systems due to limited wireless bandwidth.

Summary: We show that PRESTO scales to around hundred nodes per proxy, and can handle eight queries per minute with query drop rates of less than 5% and average latency of 3-4 seconds per query.

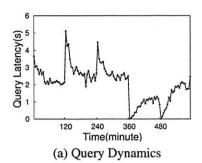
7.4 PRESTO Adaptation

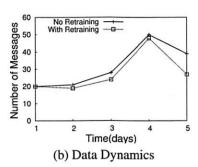
Having demonstrated the scalability and energy efficiency of PRESTO, we next evaluate its adaptation to query and data dynamics. In general, adaptation only changes what the sensor does for future data and not for past data. Our experiments evaluate adaptation for queries that request data from the recent past (one hour).

In our first experiment, we run PRESTO for 12 hours. Every two hours, we vary the mean of the distribution of query precision requirements thereby varying the query error tolerance. The proxy tracks the mean of the query distribution and notifies the sensor if the mean changes by more than a pre-defined threshold, in our case, 10. Figure 6(a) shows the adaptation to the query distribution changes. Explicit feedback from the proxy to each sensor enables the system to vary the δ corresponding to the changes in query precision requirements. From the figure, we can see that there is a spike in average query latency and the energy cost every time the query confidence requirements become tighter. This results in greater query miss rate and hence more pulls as shown in Figure 6(a). However, after a short period, the proxy provides feedback to the sensor to change the pushing threshold, which decreases the query miss rate and consequently, the average latency. The opposite effect is seen when the query precision requirements reduce, such as at the 360 minute mark in Figure 6(a). As can be seen, the query miss rate reduces dramatically since the model at the proxy is too precise. After a while, the proxy provides feedback to the sensors to increase the push threshold and to lower the push rate. A few queries result in pulls as a consequence, but the overall energy requirements of the system remains low. In comparison with a non-adaptive version of PRESTO that kept a fixed δ , our adaptive version reduces latency by more than 50%.

In our second experiment, we demonstrate the benefits of adaptation to data dynamics. PRESTO adapts to data dynamics by model retraining, as described in Section 5. We use a four day dataset, and at the end of each day, the proxy retrains the model based on the pushes from the sensor for the previous day, and provides feedback of the new model parameters to the sensor. Our result is shown in Figure 6(b). For instance, on day three, the data pattern changes considerably and the communication cost increases since the model does not follow the old patterns. However, at the end of the third day, the PRESTO proxy retrains the model and send the new parameters to the sensors. As a result, the model accuracy improves on the second day and reduces communication. The figure also shows that the model retraining reduces pushes by as much as 30% as compared to no retraining.

While most of our experiments involved the use of temperature traces as a substitute of live temperature sampling, we conducted a number of experiments with a live outdoor deployment of PRESTO using one proxy and four sensors. These experiments corroborate our findings from the trace-driven testbed experiments. The result of one such experiment is shown in Figure 6(c).





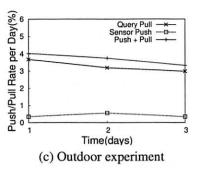


Figure 6: Adaptation in PRESTO to data and query dynamics as well as adaptation in an outdoor deployment.

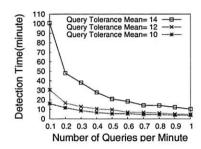


Figure 7: Evaluation of failure detection

The figure shows that, over a period of three days, as the model adapts via retraining, the frequency of pulls as well as the total frequency of pushes and pulls falls.

Summary: Feedback from the proxy enables PRESTO to adapt to both data as well as query dynamics. We demonstrate that the query-adaptive version of PRESTO reduces latency by 50%, and the data-adaptive version reduces the number of messages by as much as 30% compared to their non-adaptive counterparts.

7.5 Failure Detection

Detecting sensor failure is critical in PRESTO since the absence of pushes is assumed to indicate an accurate model. Thus, failures are detected only when the proxy sends a pull request or a feedback message to the sensor, and obtains no response or acknowledgment.

Figure 7 shows the detection latency using implicit heartbeats and random node failures. The detection latency depends on the query rate, the model precision and the precision requirements of queries. The dependence on query rate is straightforward—an increased query rate increases the number of queries triggering a pull and reduces failure detection latency. The relationship between failure detection and the model accuracy is more subtle. Model accuracy depends on two factors—the time since the last push from the sensor, and model uncertainty that captures inaccuracies in the model. As the

time period between pushes grows longer, the model can only provide progressively looser confidence bounds to queries. In addition, for highly dynamic data, model precision degrades more rapidly over time triggering a pull sooner. Hence, even queries with low precision needs may trigger a pull from the sensor. The failure detection time also reduces with increase in precision requirements of queries. For instance, for a query rate of 0.1 queries/minute, the detection latency increases from 15 minutes when queries require high precision to 100 minutes when the queries only require loose confidence bounds.

The worst-case time taken for failure detection is one day since this is the frequency with which a feedback message is transmitted from the proxy to each sensor. However, this worst-case detection time occurs only if a sensor is very rarely queried.

Summary: Our results show that sensor failure detection in PRESTO is adaptive to data dynamics and query precision needs. The PRESTO proxy can detect sensor failures within two hours in the typical case, and within a day in the worst case.

8 Related Work

In this section, we review prior work on distributed sensor data management and time-series prediction.

Sensor data management has received considerable attention in recent years. As we described in Section 1, approaches include in-network querying techniques such as Directed Diffusion [11] and Cougar [23], stream-based querying in TinyDB [13], acquisitional query processing in BBQ [3], and distributed indexing techniques such as DCS [20]. Our work differs from all these in that we intelligently split the complexity of data management between the sensor and proxy, thereby achieving longer lifetime together with low-latency query responses.

The problem of sensor data archival has also been considered in prior work. ELF [2] is a log-structured file system for local storage on flash memory that provides load leveling and Matchbox is a simple file system that

is packaged with the TinyOS distribution [10]. Our prior work, TSAR [5] addressed the problem of constructing a two-tier hierarchical storage architecture. Any of these techniques can be employed as the archival framework for the techniques that we propose in this paper.

A key component of our work is the use of ARIMA prediction models. Most relevant to our work on prediction models are the approaches proposed in BBQ [3], in which multi-variate Gaussian models were used for addressing spatial correlations, and dynamic Kalman filters for addressing temporal correlations. Our work differs in that we propose model-driven push instead of pull, and we split modeling complexity between proxy and sensor tiers rather than using only the proxy tier. ARIMA models for time-series analysis has also been studied extensively in other contexts such as Internet workloads, for instance in [9].

9 Conclusions and Future Work

This paper described PRESTO, a model-driven predictive data management architecture for hierarchical sensor networks. In contrast to existing techniques, our work makes intelligent use of proxy and sensor resources to balance the needs for low-latency, interactive querying from users with the energy optimization needs of the resource-constrained sensors. A novel aspect of our work is the extensive use of an asymmetric prediction technique, Seasonal ARIMA [1], that uses proxy resources for complex model parameter estimation, but requires only limited resources at the sensor for model checking. Our experiments showed that PRESTO yields an order of magnitude improvement in the energy required for data and query management, simultaneously building a more accurate model than other existing techniques. Also, PRESTO keeps the query latency within 3-5 seconds, even at high query rates, by intelligently exploiting the use of anticipatory pushes from sensors to build models, and explicit pulls from sensors. Finally, PRESTO adapts to changing query and data requirements by modeling query and data parameters, and providing periodic feedback to sensors. As part of future work, we plan to (i) extend our current models to other weather phenomena beyond temperature and to other domains such as traffic and activity monitoring, and (ii) design spatiotemporal models that exploit both spatial and temporal correlations between sensors to further reduce communication costs.

10 Acknowledgments

This research was supported, in part, by NSF grants EEC-0313747, CNS-0546177, CNS-052072, CNS-0520729, and EIA-0080119. We wish to thank Ning

Xu at University of Southern California for providing the James Reserve Data. Thanks also to our shepherd, Matt Welsh, as well as the anonymous reviewers for their helpful comments on this paper.

References

- [1] G. E. P. Box and G. M. Jenkins. Time Series Analysis. Prentice Hall, 1991.
- [2] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In Proc. ACM SenSys, 2004.
- [3] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. VLDB*, 2004.
- [4] P. Desnoyers, D. Ganesan, H. Li, and P. Shenoy. PRESTO: A predictive storage architecture for sensor networks. In Proc. HotOS X, 2005.
- [5] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: A two tier storage architecture using interval skip graphs. In Proc. ACM SenSys., 2005.
- [6] Emstar: Software for wireless sensor networks. http://cvs.cens. ucla.edu/emstar/.
- [7] A. Arora, et al. ExScal: Elements of an Extreme Scale Wireless Sensor Network. In The 11th International Conference on Embedded and Real-Time Computing Systems and Applications, 2005
- [8] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In Proc. ACM SenSys, 2004.
- [9] J. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. In Proc. the IEEE Intl. Conf. on Systems and Network Management, 1999.
- [10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In Proc. ASPLOS-IX, 2000.
- [11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proc. Mobicom. 2000.
- [12] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In Proc. Mobicom, 2000.
- [13] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. In ACM Transactions on Database Systems, 2005.
- [14] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In ACM International Workshop on Wireless Sensor Networks and Applications, 2002.
- [15] G. Mathur, P. Desnoyers, D. Ganesan and P. Shenoy. Ultra-low Power Data Storage for Sensor Networks. In Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS), 2006.
- [16] M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson1, D. Hahnel, D. Fox, and H. Kautz. Inferring adls from interactions with objects. *IEEE Pervasive Computing*, 3(4):50–56, 2003.
- [17] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In Proc. ACM SenSys, 2004.
- [18] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS), 2005.
- [19] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In Proc. Hot Chips 16: A Symposium on High Performance Chips, 2004.
- [20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT - a geographic hash-table for data-centric storage. In First ACM International Workshop on Wireless Sensor Networks and Applications, 2002.
- [21] Stargate platform. http://platformx.sourceforge.net/.
- [22] Center for Embedded Networked Sensing (CENS) James Reserve Data Management System. http://dms.jamesreserve.edu/.
- [23] Y. Yao and J. E. Gehrke. The cougar approach to in-network query processing in sensor networks. In Sigmod Record, 31(3), 2002.
- [24] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proc. IEEE Infocom*, 2002.
- [25] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In Proc. ACM SenSys, 2003.

Practical Data-Centric Storage

Cheng Tien Ee UC Berkeley Sylvia Ratnasamy Intel Research, Berkeley Scott Shenker ICSI & UC Berkeley

Abstract

Most data retrieval mechanisms in wireless sensor networks adopt a data-centric approach, in which data is identified directly by name rather than through the location of the node on which it is stored. Initial data-centric methods, such as directed diffusion and TinyDB/TAG, focused on the conveyance of data. One of the advantages of these algorithms is that they do not require point-to-point routing, which has proved to be difficult and costly to implement in wireless sensor networks, and instead require only the simpler and more robust tree-construction primitives.

Some recent data retrieval proposals have extended the data-centric paradigm to storage. Data-centric storage uses in-network placement of data to increase the efficiency of data retrieval in certain circumstances. Unfortunately, all such proposals have been based on point-to-point routing, and therefore have faced a significant deployment barrier.

In this paper we hope to make data-centric storage more practical by removing the need for point-to-point routing. To that end, we propose pathDCS, an approach to data-centric storage that requires only standard tree construction algorithms, a primitive already available in many real-world deployments. We describe the design and implementation of pathDCS and evaluate its performance through both high-level and packet-level simulations, as well as through experiments on a sensor testbed.

1 Introduction

Deployments of wireless sensor networks (WSNs) in recent years have grown steadily in their functionality and scale [1, 3, 13, 18, 25, 31, 34], but they still operate under extreme energy constraints. Hence, the ability to efficiently extract relevant data from within the WSN remains paramount. In their seminal paper [6], Estrin *et al.* argue that efficient data-retrieval in WSNs requires a paradigmatic shift from the Internet's node-centric style,

in which the basic communication abstraction is pointto-point (or multipoint) delivery, to a *data-centric* approach in which query and communication primitives refer to the names of sensed data rather than the identity (*e.g.*, network address) of the sensing node.

The first generation of data-centric methods addressed the *conveyance* of data through the network. Directed diffusion [14], the first such proposal, determined data routes (and rates) based on reinforcement feedback from upstream nodes, resulting in tree-like data paths from the various sensing nodes to the base station (by which we mean the source of queries). A later method, TinyDB/TAG [23, 24], explicitly constructs a delivery tree and then performs various forms of data manipulation as the data is conveyed to the base station.

A later generation of data-centric methods, inspired by the use of Distributed Hash Tables (DHTs) in the Internet, has focused on the storage rather than the conveyance of data. These solutions use intelligent innetwork storage to make data retrieval more efficient. In data-centric storage, sensed data are stored, by name, within the network. All data with the same name are stored at a single node, so queries can be routed directly (rather than flooded) to the node that stores the desired data. Data-centric storage (DCS) can be used to support a variety of sophisticated query primitives such as multidimensional range queries [11,22], multi-resolution queries [10], and approximate queries [12]. However, there has yet been any deployment in sensornets, though there are multiple scenarios in which they will be useful. For instance, we can imagine a sensor network deployed in a safari, monitoring the location of the various animals. Rather than querying each node to determine if it has seen an elephant, we can instead query a single node that is responsible for all elephant sightings.

These two classes of methods, data-centric conveyance and data-centric storage, have very different performance characteristics in terms of the energy expended to get the desired data. As discussed in [30] for the sim-

plest cases of data-centric conveyance and storage, their relative performance depends on the nature of the data generation, the query rate, the network size, and many other factors.

More to the point of this paper, these two classes of methods require very different communication primitives from the network. The various data-centric conveyance methods rely (either implicitly or explicitly) on tree-construction techniques. Note that even the simplest method of data conveyance, whereby all data are proactively sent to the base station immediately upon generation, also relies on a spanning delivery tree. Tree-based routing is both algorithmically simple and practically robust, leading to its adoption in a number of real-world deployments. For example, simple proactive data delivery was used in the deployments on Great Duck Island [25,31] and Intel's fabrication unit [3], while TinyDB is used in the deployments at the UCB botanical gardens [18].

In contrast, all known data-centric storage methods rely on a point-to-point routing primitive: they deterministically map the name (say x) of a data item to the routable address (say i) associated with a particular node. Node i is then responsible for storing all data named x and all queries for x are routed directly to node i, thereby requiring point-to-point routing.

However, as we review in the following section, achieving scalable and practical point-to-point routing is a difficult challenge. While a number of recent research efforts [8,11,17,20,27,28] have made significant progress towards this end, point-to-point routing still requires significantly more overhead and complexity than tree construction as we will explain in the following section, and has yet to be used in real-life deployments. It thus seems unwise to couple data-centric storage to such a burdensome underlying primitive, particularly one that is not widely deployed. If data-centric storage is to become more widely used, it should rely only on currently available, and easily implementable, communication primitives.

Our goal is not merely to find a better algorithm for data-centric storage. More fundamentally, we hope to make data-centric storage a basic primitive available to WSN applications, and we recognize that this can only happen if data-centric storage is implemented with minimal assumptions about the underlying infrastructure.

To that end, this paper proposes a data-centric storage method called pathDCS that uses only tree-based communication primitives. The design relies on associating data names with *paths*, not nodes, and these paths are derived from a collection of trees. We investigate some basic performance issues, such as load balance, through high level simulation but for a more real-world evaluation we implemented pathDCS in TinyOS and report

on its performance in packet-level TOSSIM [21] simulations as well as in experiments on a mote testbed. To the best of our knowledge, this is the first evaluation of a working prototype of data-centric storage. Our results show that pathDCS achieves high query success rates (on our 100-node testbed, we see roughly a 97% success rate) and is robust to node and network dynamics.

Finally, we note that in this paper we only consider the basic exact-match storage primitives as explored by schemes such as GHT [29] and GEM [27]. We leave for future work its possible extension to supporting the more complex query primitives from the literature [9, 10, 12, 22].

2 Background

The value of pathDCS relies on four basic points:

- Data-centric storage is a valuable paradigm in WSNs.
- 2. Current data-centric storage techniques rely on point-to-point routing.
- Point-to-point routing is difficult, and imposes significant overhead on WSNs.
- pathDCS provides a scalable and robust implementation of data-centric storage that does not require point-to-point routing.

The bulk of this paper is devoted to demonstrating the fourth point. In this section, we briefly review the literature supporting the first three.

Point # 1 Data-centric storage (DCS) was first explicitly proposed in [30]. Analysis of a simple model identified scenarios in which DCS outperforms the other data retrieval approaches, namely external storage (in which all sensed data is proactively sent to the base station) and data-centric routing (in which queries are flooded and only relevant data are transmitted to the base station). This same analysis also identified scenarios where the other two methods outperformed DCS. Thus, DCS and other techniques should be seen as complementary, not competitive; our assumption is that DCS is a valuable method of data retrieval in some but not all circumstances.

Reference [30] presented only the simplest form of data-centric storage: an exact-match query-by-name service where the named data can be directly retrieved. A number of subsequent proposals extend the idea of data-centric storage to support more complex queries such as multi-dimensional range queries [11,22], multi-resolution indexing [10] and spatially distributed quadtree-like indices [12].

Point # 2 Data-centric storage requires a hash-like interface where data (or data structures) can be stored and retrieved by name. In all the above proposals, this is achieved by deterministically mapping (typically by hashing) a data name to a geographic location within the network. The node geographically closest to the hashed location is deemed responsible for storing information associated with the hashed name; geographic point-to-point routing is then used to reach this storage node.

While elegant in structure, this approach requires that nodes know the network's external geographic boundary so that names are mapped to geographic locations within the network. If they don't, most data will end up being stored by edge nodes after an extensive perimeter walk, resulting in uneven load and inefficient operation. The various proposals acknowledge, but do not address, this challenge.

Point # 3 The original geographic routing algorithms such as GPSR (see [2, 16, 19]) were designed for *unit-disc* connectivity graphs under which a node hears transmissions from another node if and only if they are within a fixed radio range. (This assumption is crucial for the perimeter walk phase, but is not needed for the greedy phase of geographic routing.) Measurements have shown that this assumption is grossly violated by real radios [8, 33, 35] and that geographic routing breaks down in such cases [17].

In recent work, Kim et al. [17] and Leong et al. [20] proposed extensions to GPSR that removes the need for the unit-disc assumption. CLDP [17] represents a fundamental breakthrough in that it guarantees correct operation over topologies with even arbitrary connectivity. GDSTR [20] on the other hand routes on spanning trees when greedy forwarding is unable to make progress. In both cases additional complexity and overhead is required.

An even more basic assumption underlying geographic routing is that each node knows its geographic coordinates. While some sensor nodes are equipped with GPS, the widely-used Berkeley mote is not: although other localization techniques do exist, none of them have been evaluated for their potential to serve as routing coordinates. Motivated by this challenge, GEM [27] and NoGeo [28] explore the construction of *virtual* coordinate systems; these are synthetic coordinates to which geographic routing can be applied. Like CLDP, GEM and NoGeo represent significant conceptual advances but come at the cost of increased complexity. NoGeo requires O(N) per-node state during initialization while GEM can incur significant overhead under node and network dynamics.

Finally, there are a number of proposals for point-topoint routing in the literature on ad-hoc wireless networks. Many of these solutions face scalability problems when applied to wireless sensor networks and are thus unlikely to serve as a substrate for DCS. We refer the reader to [8] for a more detailed discussion of the space of point-to-point routing algorithms and their applicability to WSNs.

As the above discussion reveals, there has been significant progress on point-to-point routing for WSNs and both BVR and CLDP have resulted in working implementations for the mote platform. At the same time, the various solutions remain fairly complex (at least relative to tree construction) and face further challenges in supporting in-network storage. For these reasons, we deemed it worthwhile to explore an alternate approach that releases DCS from the challenges and complexities of point-to-point routing.

3 Design

We begin this section with the description of the core pathDCS algorithm, followed by those of supporting ones.

3.1 Core Algorithm

For pathDCS to be effective, it must be consistent: that is, all queries and stores for the same object (no matter from where they are issued) must reach the same destination. The traditional way to ensure consistency is to give all nodes a shared frame of reference that allows packets to describe their destination and enables forwarding nodes to route packets to that destination. We use a few shared points of reference called landmarks (places with well-known names that all nodes can reach), and name locations by their path from one of these shared points of reference [32]. For example, when giving driving directions (in real life) we often use a well-known landmark and then provide path-based instructions: "Go to the gas station, and then take your first right, and then after two blocks take a left...." The driver need only know (a) how to find the landmarks and (b) how to follow a set of procedural directions. This is the approach used in pathDCS. We map each name to a path, not a node, and that path is defined by an initial landmark and a set of procedural directions that are defined in terms of other landmarks. To query or store that name, a packet goes to the designated landmark and then follows a set of procedural directions; the store or query is then executed at the node on which the path ends. Notice that the endpoint of the path is independent of where the query or store is issued from; since the path starts off by going to a particular landmark, its origin doesn't matter.

In pathDCS the landmarks are a set of beacon nodes, which can be elected randomly or manually configured (see Section 3.2). To make sure that all nodes know how to reach the beacons, we use standard tree-construction

techniques to build trees rooted at each one of these beacons. The overhead to establish the necessary state is proportional to the number of beacons; as we will see, that number is small so pathDCS imposes little overhead.

The paths are specified in terms of an initial beacon and a set of segments, with each segment consisting of a direction (defined in terms of a beacon) and a length (defined by how many hops). Thus, each path consists of a sequence of p beacons b_i and lengths l_i , where $i=1,\ldots,p.^1$ The packet is first sent to beacon b_1 . From there, it is sent l_2 hops towards beacon b_2 using the tree rooted at b_2 . The process then repeats; from wherever the packet ended up at the previous i-1 segment, it is then sent l_i hops towards the next beacon b_i . The path ends after the pth segment.

To make this more precise, we first define some terms. There is a linear space of identifiers, say 16-bit addresses, that is large enough so that there are no clashes in identifier assignments. Each node in the network is assigned a logical identifier id. Data is associated with a key k (assume this is derived from a hash of its name) and, for node n, the hop distance to beacon b is given by hops(n,b). Let n_i denote the identifier of the node on which the ith segments starts (also the place where the previous segment ends). Lastly, there is some hash function h(k,i) which maps an identifier k and an integer i into an identifier.

When accessing a data item with identifier k, the set of beacons used for the path are determined by consistent hashing [15]: beacon b_i is the beacon whose identifier is *closest to* (in the sense of consistent hashing) the identifier h(k, i). In addition, the first segment length l_1 is always equal to the distance to the first beacon b_1 , whereas segment lengths for i > 1 are given by:

$$l_i = h(k, i) \bmod hops(n_i, b_i) \tag{1}$$

We use Figure 1 as an example to illustrate how pathDCS routes packets with the same key from different source nodes to the same destination node. For clarity we show the routing trees rooted at b_1 , b_2 and b_3 in Figures 1a, 1b and 1c respectively. We fix the total number of path segments at 3, and both source nodes s_1 and s_2 generate packets with the same key k. Both the current number of remaining hops and the current path segment i (also called *segment counter*) are carried in the packet header and modified as needed. In the figure, beacons b_1 , b_2 and b_3 are chosen because their ids are closest to h(k,1), h(k,2) and h(k,3) respectively. The order of beacons towards which packets are forwarded is

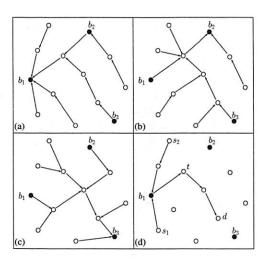


Figure 1: (a), (b) and (c) show the routing trees rooted at beacons b_1 , b_2 and b_3 respectively. (d) Source nodes s_1 and s_2 both send packets with the same key. These packets first reach a common first beacon (b_1) , before taking the same subsequent path segments to reach destination node d.

therefore b_1 , b_2 and b_3 , following the order of segments traversed. Initially, both packets are routed to b_1 , upon which it is determined, using Equation 1, that in the second segment they should be forwarded towards b_2 for, say, 1 hop. At node t, which is the terminating node of the second segment, the segment counter in the packet header is incremented, the number of hops is again computed using Equation 1 (assume the result is two), and the packets are subsequently forwarded two hops towards the third and final beacon, to terminate at node d. Node d is then the destination node for all data associated with key k.

The number of hops required for each query or store is proportional to the diameter of the network, which is the same for all DCS approaches, multiplied by the number of segments. Thus, the key to keeping the overhead of pathDCS manageable is keeping the number of segments, p, small. As we argue below, p=2 is sufficient for reasonably-sized networks, so that we expect the perquery expenditure to be a few multiples bigger than in other DCS schemes.

The pathDCS algorithm has two parameters: B, the total number of beacons and p, the number of path segments. Varying B trades off the control traffic overhead due to tree construction versus load on the beacons. We explore this tradeoff in Section 5. With regards to p, increasing the number of segments results in longer paths but potentially spreads the storage load more evenly. To see this how large p should be to achieve reasonable load distribution, consider the following naive back-of-the-envelope calculation. The total number of paths a message can traverse using pathDCS is approximately B^p . Letting d be the network density and r the radio range,

¹Note that the labeling of the beacons b_i is idiosyncratic to a path; that is, the indices i merely refer to the ordering of beacons in this particular path. We don't introduce a notation for an absolute labeling of the beacons.

the expected length of each path is given by

$$\frac{1}{2r}\sqrt{\frac{N}{d}}\tag{2}$$

Thus the number of nodes pathDCS routing can potentially use to store data is approximately

$$\frac{B^p}{2r}\sqrt{\frac{N}{d}}\tag{3}$$

Equating 3 to total number of nodes N, the number of beacons required is given by

$$\left(2r\sqrt{dN}\right)^{\frac{1}{p}}\tag{4}$$

As an example, we plug in the following values: r=8 units, d=0.07 nodes per unit area, N=20000, and for p=2, we obtain $B\approx 24$, which is a reasonable number. We did simulations for p=2,3,4,5 to verify that indeed the distribution of load changes very little with increasing p and then picked p=2 since it, as expected, resulted in the shortest paths. Note that knowledge of N by every node is not required, only p and p need be set at deployment. Unless the network size changes drastically we do not expect performance to degrade significantly.

3.2 Supporting Algorithms

While the basic idea of pathDCS is contained in the core algorithm defined above, actual implementation of pathDCS requires a set of supporting algorithms to, for example, select beacons and build trees. There is nothing novel in these algorithms, we describe them for completeness.

Tree Construction To construct a tree rooted at a particular beacon, we recursively have nodes pick a parent that is closest to that beacon amongst all their neighbors. Our implementation uses the ETX [5], also the MT [33] metric as an indication of path quality.

Beacon Election The total number of beacons in the system is a fixed constant B, and is dependent on the size of the network. We divide the identifier space into B equal partitions, and have each node compete to become the beacon for the partition in which they reside. Borrowing the basic concept from SRM [7], each node's self-election announcement is delayed by time proportional to the difference between their ids and the largest identifier for that partition (i.e. the identifier that describes the upper boundary of that partition). For instance, if we assume that B=4, and node X, Y and Z's identifiers fall within the partitions 2, 2 and 4 respectively, only X and Y compete to be the beacon in partition 2. X and

Y independently set a timer with delay $\alpha(I_2-id_X)$ and $\alpha(I_2-id_Y)$ respectively, where I_2 is the largest possible identifier for that partition, and α is some constant. This scheme ensures that node Y, with the higher id, usually announces itself before X, thereby suppressing X's announcement.

It is possible that the election process results in two or more beacons clustering. An additional rule can be imposed to reduce the occurrence of this scenario: when timeout occurs and just before a node announces itself as a beacon, it checks to see if any beacons lie within k hops. If so, it suppresses its announcement.

Beacon Handoff and Failure From time to time, the role of beacons should be handed over to other nodes, either due to failures, or to reduce the forwarding load on the beacons. In the case of the former, one hop neighbors begin a self-election process once the link quality to that beacon drops below a threshold. Similar to the initial election process, the delay for the timer set is a function of the difference between the current node's identifier, and that of the highest identifier for that partition. Note that in this case all one-hop neighbors participate in the election. The winning node then takes over the identifier of the deceased beacon, and assumes that role henceforth. For the case of deliberate handoff, the beacon randomly picks a neighbor, and switches identifiers with it. Possible different criteria exist, the meeting of any one can trigger deliberate handoff. An example of a criterion would be a minimum amount of remaining energy. In this case, the time at which handoff is triggered is very much dependent on the rate at which the application generates data packets. One can also imagine the beacons handing off in order to spread themselves out if they are clustered together. The proximity of the current and previous beacon ensures that drastic route updates in the network are minimized. Specifically, the destination nodes for a particular key before and after the handoff takes place should lie close to each other, in terms of number of hops. Together with data replication mentioned below, this increases the chances of finding the data before the next data refresh (see below) or before new data is stored at the updated location.

Responding to Queries In the typical case, where the querying node is the base station (or any other well-defined node), we construct a tree rooted at that node. Answers to queries are sent back along this tree to the base station. If queries are being issued from multiple nodes, then each such node includes its closest beacon in the query. Backward path establishment from that beacon is performed by storing pointers to the previous node at each intermediate hop. Responses to queries are sent back to the closest beacon (as noted in the query) and that beacon forwards the response along the path that was es-

²resulting in an average of 14 neighbors

tablished from the querying node by the path establishment message.

Data Refreshing Every node where data is stored will periodically issue *refresh* probes for those data. These probes are routed in the same manner as the data packets, allowing the node to detect if the topology has changed since the initial storing. If the node initiating the refresh does not receive the probe in return, it then stores the data at the new location.

Data Replication Finally, local replication of data is performed at the storage node. Data packets are disseminated using a localized flood within k-hops of the destination. A query reaching a destination not storing the required data is similarly flooded locally, with replication nodes responding to the query.

4 Performance Metrics

Before proceeding to the sections on simulation and implementation details and results, we elaborate on the metrics of interest, namely path consistency, storage and forwarding load balance.

The design of pathDCS raises three performance questions. The first has to do with the consistency with which pathDCS maps names to storage locations. In the absence of node and network dynamics, pathDCS achieves perfect consistency in that stores and lookups for a particular data item always terminate at the same storage node, and hence pathDCS would see a 100% success rate for lookups. However, node and network dynamics can lead to changes in the paths to beacons and hence to changes in the mapping between a name and storage node. The extent to which such changes impact lookups depends on both the frequency and the extent of changes. If changes in storage nodes are highly localized, then simple local replication of data should trivially mask such changes. If changes are infrequent, then a periodic refresh of stored data should suffice to maintain high success rates. In any case, pathDCS provides only weak consistency: it does not guarantee that the data retrieved is the latest stored.

These path changes are primarily dependent on the behavior of the wireless medium and hence we explore this issue in detail in Section 6. However, such changes are also dependent on network size because longer paths are more likely to experience changes. Since we can't analyze scaling effects on our testbed, we use an idealized, but highly pessimistic, model of path dynamics in our simulator to see how consistency varies with system size. To quantify consistency, we measure the *lookup success rate*, which is the probability that a lookup for a data item *x* reaches a storage node currently storing *x*. To understand the magnitude of lookup variations, we also measure the maximum separation in hops between any two

nodes storing a particular data item, which we call the *spread*. This measures the extent to which local replication can mask the effect of path dynamics.

The second performance issue has to do with how effectively pathDCS balances the storage and forwarding load across nodes. This is a potential issue because unlike other DCS schemes that explicitly distribute data items over the set of all nodes, pathDCS distributes data over a more limited number of paths. While we do not expect pathDCS to achieve load distribution comparable to the address-based DCS schemes, we would like to verify that the load distribution in pathDCS is not unreasonable.

5 High-Level Simulation Results

5.1 Overview

The performance of pathDCS derives from the inherent behavior of its algorithms as well as the impact of the wireless medium on both the algorithms and our particular implementation choices. To separate the effects of each, we evaluate pathDCS through a combination of high-level simulations (to evaluate the scaling behavior of the algorithms themselves), low-level simulations that take into account a lossy medium and packet collision effects, and implementation (to evaluate pathDCS under realistic wireless conditions). This section presents our high-level simulation results; our prototype and its evaluation in TOSSIM [21] and on actual testbeds are described in Section 6.

Our simulator makes a number of simplifying assumptions that abstract away the vagaries of the wireless medium. Nodes are placed uniformly at random in a square plane and every node is assigned a fixed circular radio range. A node can communicate with all and only those nodes that fall within its radio range. In addition, the simulator does not model network congestion or packet loss. While clearly unrealistic, these simplifications allow simulations that scale to thousands of nodes; our packet-level simulation and testbed results in the following section capture performance under more realistic conditions.

Our default simulation scenario uses 5000 nodes placed in an area of 6.7×10^4 units² with a radio range of 8 units, leading to an average node degree of 14.5. We maintain the same density for all simulations.

5.2 Lookup Success Rates

The path to a beacon can change for two reasons: (1) tree reconfiguration following node failure(s) and (2) variations in link quality that trigger a change in a node's choice of parent. The first we can accurately model in simulation, the second we can only roughly approximate.

We perform the following test to measure success

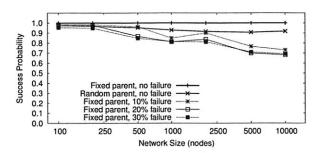


Figure 2: Success rate under failure and randomized parent selection for increasing network sizes. All tests use 20 beacons and 2 path segments.

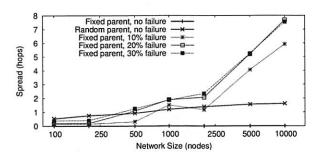


Figure 3: Spread under failure and randomized parent selection for increasing network sizes. All tests use 20 beacons and 2 path segments.

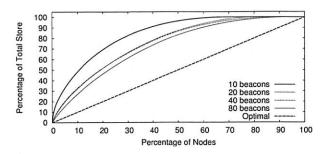


Figure 4: CDF of storage load with pathDCS and "optimal" DCS for different numbers of beacons.

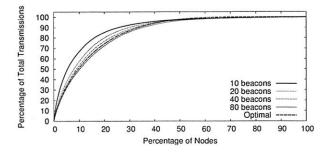


Figure 5: CDF of transmission load with pathDCS and "optimal" DCS for different numbers of beacons.

rates: for a network with N nodes, every node inserts

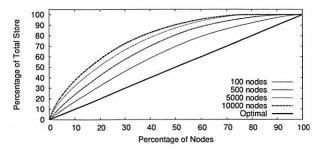


Figure 6: CDF of storage load using pathDCS for increasing network sizes.

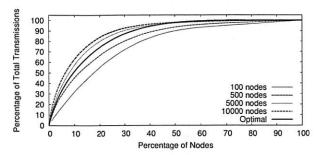


Figure 7: CDF of transmission load using pathDCS for increasing network sizes.

10 distinct data items into the network yielding a total of $10 \times N$ distinct data items. Stores are replicated within the one-hop neighborhood of the destination. We then perform 20 lookups for each data item. A lookup succeeds if it arrives at a node storing the requested data item (either at the original destination, or at one of the one-hop replicas); otherwise, the lookup fails. To measure spread, we repeat the same tests as above but now we turn off one-hop replication and have each node store (rather than lookup) every data item 20 times. For each data item, we then consider the set of nodes storing that item and measure spread as the maximum separation in hops between any two nodes in the set.

To capture the effect of node failure, after each of the 20 iterations for a given item, we fail a fixed fraction of nodes uniformly at random and then recompute the trees for each beacon. Capturing varying link qualities is more problematic because our simulator does not model congestion and packet loss; instead, we directly address the effect on parent selection. We conservatively model changes in parent selection arising from varying link qualities as follows: rather than pick a single parent for each beacon, a node considers *all* of its neighbors that are closer to the destination beacon than itself as potential parents. For every message, a node then chooses its next hop uniformly at random from this entire set of potential parents. This represents a highly pessimistic scenario in which, at every hop, the route to a beacon

can flap between all possible next-hops.³

Recall that fixed parent selection with no failure has a success rate of 100% and a spread of zero since we turn off one-hop replication when measuring spread. Figures 2 and 3 plot the average success rate and spread under increasing network size using random parent selection or fixed parent selection under various failure rates. As expected, we see that the success rate drops, and spread rises with network size but this deterioration is slow. For example, a 10,000 node network with random parent selection (which, again, is a pessimistic model) still sees a success rate of 92%. Moreover, the absolute value of spread is often low and hence could frequently be masked by simple k-hop local replication. We implement just 1-hop replication but for very large networks (>10,000 nodes) with high failure rates (\sim 30%) one might need a larger scope of replication.

Section 6 continues this evaluation in real wireless environments.

5.3 Load Distribution

There are only two knobs to the basic pathDCS algorithm: (1) the total number of beacons and (2) the number of path segments used. Ideally, we want to pick a number of beacons and path segments that allow forwarding and storage load to be well spread out while maintaining reasonable path stretch. The analysis in Section 3 leads us to the choice of 2 segments and hence we now look at the number of beacons required to achieve good load distribution.

We first hold N, the network size, fixed at 5000 nodes and scale B, the number of beacon nodes. As before, every node uses pathDCS to insert 10 distinct data items into the network yielding a total of 50,000 distinct stored items. We then measure the per-node forwarding and storage load. Figures 4 and 5 plot the cumulative distribution function (CDF) of the storage and transmission load respectively. To determine if any load imbalances are due to pathDCS, or are inherent in the DCS approach, we also plot the distributions for an "optimal" form of DCS in which names are mapped uniformly at random over the entire set of nodes and stores follow the shortest path from the inserting node to the destination storage node.4 In terms of storage, we see that usage of just 20 beacons results in a fairly even distribution and that increasing B beyond 20 offers rapidly diminishing returns. In terms of transmission load, we see that the pathDCS distribution approaches that of the optimal although both are fairly skewed. This is due to the natural concentration of traffic in the center of the grid and is in no way specific to pathDCS or even DCS schemes in general; rather this is an issue for communication in all ad hoc networks and one that has received some attention in the literature [26].

At less than 1% of the total number of nodes, B=20 represents very low control overhead in terms of tree construction. Moreover, we see that the pathDCS distributions are reasonably close to the optimal node-based DCS. Given the relative simplicity of pathDCS, this seems like a very worthwhile tradeoff.

We now investigate the variation of performance with increasing network size. We fix B=20 and scale N. Figures 6 and 7 plot the CDF of transmission and storage load respectively. We see that, as expected, the distribution deteriorates with increasing N but this deterioration is very gradual.

Finally, the stretch in all our tests was approximately 2.4 which is in keeping with our use of 2 path segments. We also verified that stretch increases as we increase the number of path segments.

In summary, this section explored the basic scaling behavior of the pathDCS algorithms. We show that pathDCS is robust in that it achieves high success rates under highly pessimistic models of node and network dynamism. Moreover, pathDCS is scalable in that it requires a small number of beacons to achieve good load distribution.

6 Implementation Details and Results

We implemented pathDCS in TinyOS, and evaluated its performance on the 100-node Intel Mirage [4] micaZ testbed as well as on 500 nodes in TOSSIM's packet-level emulator. We begin this section by briefly describing the pathDCS system architecture, followed by low-level details of the implementation in TinyOS, and finally ending with evaluation of its performance.

6.1 PathDCS System Architecture

Figure 8 shows the pathDCS system architecture, which can be divided into control and data planes. The control plane provides primarily beacon election and tree-based routing capability, whereas the data plane implements the core pathDCS forwarding logic (using the control plane's tree-based routing tables), storage of name-value pairs, and one-hop replication. Note that the only component specific to pathDCS is the forwarding engine; the remaining components are common to a number of other systems such as TinyDB [24] and BVR [8].

6.2 Control Plane

We next elaborate on the implementation of the control plane. This component primarily constructs trees rooted

³Note that we restrict parent changes to those that are localized in that they do not trigger a re-computation of the tree downstream from the changing node. The effects of non-localized changes are captured by the tests for node failure.

⁴In practice, implementing this form of optimal DCS would require every node to have global knowledge of all the nodes in the system as well as shortest path point-to-point routing.

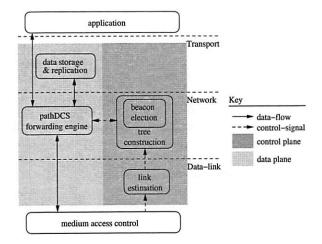


Figure 8: The pathDCS architecture is made up of (1) the data plane, consisting of the forwarding engine and data storage and replication component, and (2) the control plane, consisting of the tree construction and related components.

at the beacons, disseminating and maintaining information used to determine the next hop at every node for each beacon in the network. We begin by describing the network-wide naming mechanism.

Node Identifier Each node in the network is assigned a hardware address⁵ that is unique in the sensornet. This address is subsequently hashed to obtain the corresponding network identifiers. Since the hash function is known, collisions can be avoided by setting the hardware addresses appropriately.

Beacon Election Periodically, each node broadcasts distance vector packets containing the identifiers and distances to each beacon in the network. If the relevant element in the vector indicates a beacon in the same partition with a smaller identifier, a node elects itself by replacing the identifier with its own before broadcasting. In this manner, the entire network eventually learns the beacons and their corresponding identifiers.

Link Quality Estimation Nodes periodically broadcast estimates of their neighbors' reception qualities within their immediate neighborhood, allowing these neighbors to compute the link quality in both directions, thus accounting for asymmetric links. Messages are jittered slightly at each node to minimize interference. The link estimator module maintains, for each neighbor, a periodically updated estimate of link quality, which is the expected number of transmissions to that neighbor. This is computed as an exponentially weighted moving average:

$$L_{av,i} = (1 - \alpha)L_{av,i-1} + \alpha L_i$$

where $L_{av,i}$ and L_i are the average and sample respectively for iteration i and α is a constant. Only route update packets are used as samples for the computation.

Tree Construction Beacons periodically generate routing updates which are propagated through the network in a distance vector manner. A node uses the beacon election rules described in Section 3 to decide whether it should elect itself as a beacon. Nodes use these route updates to compute their minimum "distance" to each beacon. To reduce transmission loss, we use the MT [5], or ETX [33] metric, where the number of expected transmissions to each beacon is minimized.

Control packets and fields Control messages broadcasted by a node include information such as its current hop distance and estimate of the expected number of transmissions to each beacon, the latest sequence numbers associated with the corresponding beacon, and the node's estimate of its neighbors' reception qualities. To remove the occurrence of one-hop count-to-infinity problems, control packets also include the next hops for each beacon, so that a node does not attempt to forward packets to its neighbor which will subsequently forward the packet back.

6.3 Data Plane

In this paper, the data plane operations of interest include the forwarding of pathDCS data packets and their replication. The description of these operations is followed by an brief coverage of the packet header overhead.

Forwarding Packet headers include fields that contain the key and the current path segment the packet is on. Based on routing information provided by the control plane, these are used to determine the next beacon to route towards and the number of hops to take, as elaborated in Section 3.1. The remaining hops before reaching the end of a segment is also carried in the header.

Replication Replication of data to improve availability is achieved by scoped flooding once the data packet reaches its destination node. The field previously used to indicate the number of remaining hops is used to specify the scope of the flood, and is decremented with each subsequent broadcast. To prevent repeated broadcasting of the same packet in the local neighborhood of a node, a cache of the most recently sent ones is kept.

Data packets and fields The overhead incurred in each data packet is small. In our implementation, pathDCS uses 6 bits to represent the key associated with each data type, thus allowing for a total of 64 keys.⁶ In general we expect the number of unique data types to be small and independent of the size of the network. Also,

⁵The TOS_LOCAL_ADDRESS in TinyOS.

⁶To accomodate more keys we can simply use more bits.

in the case where the number of path segments is 2, we require an additional bit to keep track of the current segment the packet is on. Finally, the remaining hops to the terminating node of the current segment is also stored in the header, and is on the order of O(logD), where D is the maximum diameter of the network. In our implementation, the total number of control bits used to route data is just (data + segment + hops) = 6 + 2 + 8 = 16 bits.

6.4 Methodology

The primary concern when implementing pathDCS is the impact of its dependence on path stability. Whilst the construction of routing trees had been studied extensively, the main focus in previous studies was the successful forwarding of packets to the destination. Of little or no significance were the paths along which data packets traverse as long as they can get there. In pathDCS, the destination node is effectively defined as a function of the network's current routing state. As a result, if the network state changes frequently, we may store and query data items at destinations that shift rapidly with time. Such a situation will result in poor lookup success rate, rendering pathDCS less useful. This is therefore the most important point to address in our implementation.

Thus, as in Section 5, we are primarily interested in the probability of lookup success. A lookup can fail either because the message was dropped along the way, or because the destination node it arrived at did not have the requested data. Two metrics are used to distinguish between these two causes. The first is the route completion probability, measured as the probability that a packet successfully arrives at a destination node (as opposed to being dropped along the path). Note that the route completion probability has little to do with the pathDCS algorithms per se. Instead, such failures are dependent primarily on the characteristics of the wireless medium and our implementation choices for the link estimation, tree construction and retransmission modules. In general the quality of links in the network fluctuates over time, resulting in route updates as the network attempts to pick routes that result in lower loss.

The second performance metric is our usual **lookup** success rate as defined in Section 4. In computing this rate, we consider only those lookups that complete (that is, they reached some destination node), and we say that a lookup is successful if it locates the requested data item. To measure the effect of variable node and network conditions, we obtained the lookup success rate for different values of data refresh intervals. This is achieved as follows: in each experiment, we have all nodes periodically route some number of messages for each distinct data item. For the routes that do complete, we then observe where those messages terminate. Next, we divide time

into windows, where the first data packet in that window is treated as a store or refresh packet, and the node at which it terminates is the storage node for that window. Subsequent packets then act as queries and lookup success is measured as the probability that a lookup arrives within the one-hop neighborhood⁷ of the storage node for that window. We do this for each distinct data item, compute the average lookup success and repeat for different window sizes. We note that varying this window size is equivalent to altering the data refresh interval, and we can thus use a single experiment to observe the effect of increasing refresh intervals rather than running repeated experiments that may suffer from different time-of-day effects.

Data refreshing plays a crucial role in improving lookup success, especially in large networks (of size in the hundreds to thousands), where the path may vary widely over time. When we consider short time-scales, say within a period of a minute or two, the set of destination nodes for a particular key is probably small in number, and not likely to be spread out over a large region. However, when looking at all possible destinations over a period of a day, the set of nodes will be the union of all sets at shorter time-scales: it is more likely to be large, as well as covering a wider area. Thus, a refresh rate that is high translates into observation at small timescales, which means that destinations are close together, and therefore lookup success increases. We validate this in the following sections, via simulation and experiments on the testbed.

6.5 TOSSIM Simulations

In this section we describe packet-level simulations that model a lossy medium. A total of 500 nodes were simulated using actual TinyOS code. We begin by elaborating on the parameters used in the simulations as well as in the testbed's motes.

Control plane parameters The appropriate choice of parameters is dependent on the size of the network, as well as the desired properties of the applications that run on it. For instance, as we shall demonstrate in the subsequent sections, stable paths are a prerequisite for high lookup success in pathDCS. Stability can be improved by damping route updates at the expense of increased network reaction time to topology changes. Our choice of system parameters is shown in Table 1, and has been experimentally verified to yield satisfactory performance.

Data plane parameters The heart of pathDCS lies in the data plane. Parameters associated with pathDCS can be tuned here, and are largely independent of the control plane. As in Section 5, we use only 2 segments in

⁷This reflects the local one-hop replication.

Table 1: Control plane parameters

Parameter Description	Value	
Number of beacons	5	
Distance vector (DV) broadcast interval	10 seconds	
Maximum DV broadcast jitter	1 second	
Frequency of route updates	1 per 10 DV pkts	
Maximum entries in neighbor table	16	
Moving average for link estimation, α	0.05	

Table 2: Data plane parameters

Parameter Description	
Number of path sections	2
Scope of flooding for local replication	1
Maximum retransmissions to the next hop	
Maximum cached replication entries	

our implementation. This leads to lower stretch, an important consideration in real networks since longer routes result in an increase in loss probability. A shorter route is important also because it results in fewer transmissions, which consume energy in resource constrained sensor motes. Table 2 shows the parameters used in the data plane.

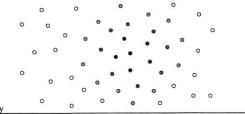
In each experiment, every node in the network generates different data items, thus data destined for a particular destination node originate from multiple sources. On average, data packets are injected into the network at the rate of two per second to minimize congestion. A total of 20 experiments are run, with each experiment querying or storing a particular, distinct key, and each node sending 72 packets. The network is allowed to run for a simulation hour for routing to converge before queries and storage began.

With the above parameters, we measure the route completion probability, the destination node spread distribution, and the lookup success rate under two test scenarios:

Normal We measure performance using the above default parameter selection,

Fast Route Adaptation We look at the impact of our choice of parameter selection on path stability and consequently on pathDCS performance. Specifically, the parameters for this test are the same as those for "Normal" except that the DV broadcast interval in Table 1 is reduced to 5 seconds, and the corresponding maximum jitter to 0.5 seconds. In general faster route adaptation can be desirable for reduced recovery time from failures.

Since paths are likely to change over time, we need to investigate the destination spread distribution. Even though we expect the spread of destination nodes to be



- node with the most packets (mode) node 2 hops away from mode node other node
- node 1 hop away from mode node ⑤ node 3 hops away from mode node

Figure 9: The measure of destination spread distribution is based on the hop distance from the node that received the most packets (i.e. the mode).

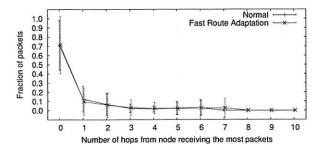


Figure 10: 500-node simulation: distribution of destinations for the normal and fast route adaptation scenarios.

significant, the situation is not hopeless if we find that most of the packets still fall within a small region. Using Figure 9 as an illustration, we proceed as follows: we first determine the node that received the most number of packets for a particular key, we call this the *mode node*. Then, for each hop from the mode node, we compute the total fraction of packets that end on nodes at that distance. If the destination nodes are evenly spread out over a wide area, then the distribution will show a relatively flat graph over multiple hops. On the other hand, if the nodes are highly clustered together, we should see a graph that peaks at the 0th hop, with small fractions at the other hops.

Observations In both scenarios, the mean network diameter is 18, the average probability of route completion is about 86%, and the mean number of neighbors is around 10.4. Figure 10 shows the distribution of destination nodes, from which we can observe the following:

 The majority of packets (~80%) land within one hop of the mode node. This implies that, without data refreshing and with one hop replication, the mean probability of lookup success will also be about 80%. As we shall see subsequently, data refreshing increases this probability. Another alterna-

⁸Since the network diameter is large, we expect the end-to-end loss probability to become significant, even with the use of link-level retransmissions. Thus, this does not reflect on pathDCS, only the underlying packet loss behavior.

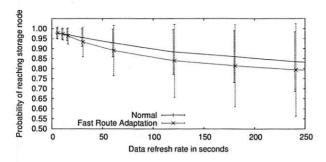


Figure 11: 500-node simulation: variation of lookup success with data refresh interval.

tive will be to increase the scope of local replication, which will however be at the expense of more transmissions and storage space.

2. Having more dynamic routing does not affect the resulting destination spread. This is due to the fact that, over time, all possible combinations of routing state, and correspondingly all possible destinations, have been encountered. Increasing the rate at which changes occur does not affect this destination set.

We next consider the effect of data refreshing. As described in Section 6.4, lookup success now refers to average fraction of queries that end within the replication region for all window periods. These periods, or refresh intervals, are varied from 5 to 250 seconds, and the results are shown in Figure 11. We can observe that

- 1. Refreshing data more frequently can increase the probability of a successful query to >95%.
- Faster route adaptation results in lower lookup success for a particular refresh interval.
- 3. Variation in lookup success is higher for routing that updates more frequently.
- 4. As the refresh interval increases, lookup success probability approaches that of packet fraction received within one hop of the mode node, which agrees with Figure 10.

Overhead Scaling Finally, we consider the overhead incurred by pathDCS, focusing on the total number of each type of packet transmitted. We identify five types of packets: (1) distance vector (DV), (2) link estimation, (3) data packet transmission for replication, (4) data packet transmission for forwarding, and (5) data packet transmission for refreshing. Figure 12 shows the breakdown for various application data generation rates. We assume that the refresh per data type occurs once every 100 seconds, and that there are 100 data types in the network. The rest of the network parameters are as given in Tables 1 and 2. From the figure, we see that the fraction of

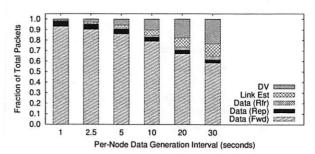


Figure 12: 500-node simulation: breakdown of transmissions for each packet type.

overhead packets reduces with an increase in application rate, which is what we expect in general. Furthermore, the cost of refreshing data is low, compared to the initial data replication and forwarding.

To summarize, the 500-node packet-level simulation shows that local replication by itself is sufficient to result in high (80%) lookup success. Refreshing data periodically counters the effects of routing changes, and is able to increase lookup success to (>95%). However, the tradeoff is that more packets are transmitted, increasing the overhead incurred.

We now proceed to evaluate the performance of pathDCS on the Intel Mirage testbed.

6.6 Testbed Details and Results

The Mirage testbed is located indoors, covering an entire floor of the building. The 100 micaZ motes are spread out over an area of approximately 160' by 40', at the locations indicated in Figure 13. Each mote in the testbed has a serial interface that is connected to an internal ethernetwork, which in turn is accessible via the Mirage server. Binaries are uploaded and data downloaded via this ethernetwork, with the server providing timestamping service for downloaded data packets. We reduce the transmission power of the motes to reduce the effective density of the network. For all our experimental results in this section, the diameter of the network is 6. Packet generation, test scenarios and network parameters are the same as that of the packet-level simulations in Section 6.5.

Results For the testbed, the mean number of per node neighbors is about 11.8, with the probability of route completion being 97.9% and 96.1% for the *normal* and *fast route adaptation* tests respectively. Figure 14 shows the spread of the destination nodes for both test scenarios. We see that in both cases the majority of packets terminate at a particular destination node, 87% for *normal* and 93% for *fast route adaptation*. If we consider all packets that terminate within one hop of the mode node, this figure rises to about 97% in both cases. Note that this takes into account all possible route changes and

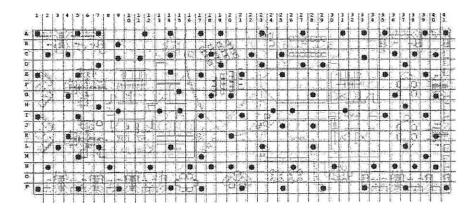


Figure 13: Location of sensor motes of the Intel Mirage testbed is indicated by the stars.

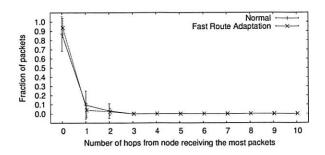


Figure 14: Distribution, or spread, of the destination node. The fraction of packets landing x-hops away from the node with the highest fraction is shown.

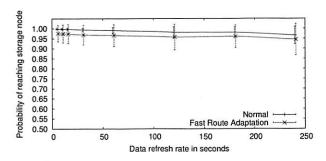


Figure 15: Probability of lookup success for particular data refresh intervals.

thus destinations for the duration of an experiment, and does not include the benefits gained from data refreshing. Thus, it is clear that for the testbed of size 100, we can obtain high lookup success even without refreshing.

On the other hand, when we consider data refresh, a routing system that is more dampened increases the chances of lookup success. Figure 15 shows the corresponding lookup success probabilities for these two systems. Four observations can be made from the figure:

1. The lookup success is very high in both cases even with low refresh rates, which agrees with the observation in Figure 14 that the set of possible destina-

tion nodes is small.

- With increased damping, the system, in particular the paths, are more stable, resulting in less variation in lookup success.
- 3. For a given refresh rate, lookup is generally worse for a more adaptive control plane.
- pathDCS constructed over a more dynamic routing control plane has to refresh its stored data more frequently in order to meet more stringent lookup success requirements.

In conclusion, the performance of pathDCS ultimately relies on the choice of parameters at the underlying control plane. Although the instability of paths causes the set of destination nodes to increase, we find that in general they tend to be highly clustered, with the majority of packets terminating on a small subset. Thus path fluctuations can be countered via two mechanisms: an increase in the scope of local replication, or an increase in the frequency of data refreshes. The former trades off storage space and additional transmissions for an increase in lookup success, whereas the latter trades off additional transmissions. From our results we believe that pathDCS is a feasible and simple way to implement data-centric storage in WSNs.

7 Summary

This paper describes a new approach to implementing data-centric storage (DCS). Our goal was not merely to find a new DCS algorithm, but to develop a more practical approach to DCS, one that does not rely on point-to-point routing. While point-to-point routing may one day be ubiquitously available in WSNs, it is not widely available now and current implementations are either based on idealized radio behavior or incur significant overhead and complexity. In contrast, tree construction primitives are widely available, and are becoming a rather standard

component in most WSN deployments. Thus, DCS has a far better chance to become a basic and widely deployed WSN primitive if it only depends on tree-based routing.

From simulations and actual deployment, we see that the primary obstacle, namely fluctuating paths, can be overcome via the usage of local replication and data refreshing. Although these two mechanisms are not perfect in that they incur additional overhead, nonetheless they perform well enough for pathDCS to be of use in large WSNs.

References

- ALLEN, M., AND ET AL. Habitat sensing at the James San Jacinto Mountains Reserve. Tech. rep., CENS, March 2003.
- [2] BOSE, P., MORIN, P., STOJMENOVIC, I., AND URRUTIA, J. Routing with guaranteed delivery in ad hoc wireless networks.
- [3] BUONADONNA, P., GAY, D., HELLERSTEIN, J. M., HONG, W., AND MADDEN, S. TASK: Sensor network in a box. In European Workshop on Sensor Networks EWSN (2005).
- [4] CHUN, B., AND BUONADONNA, P. Intel mirage testbed. Tech. rep., Intel, 2005.
- [5] COUTO, D. D., AGUAYO, D., BICKET, J., AND MORRIS, R. A high-throughput path metric for multi-hop wireless networks. In *Proceedings of the 9th Annual MOBICOM* (2003), ACM Press.
- [6] ESTRIN, D., GOVINDAN, R., HEIDEMANN, J., AND KUMAR, S. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual MOBICOM* (1999), ACM Press.
- [7] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C., AND .ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM* (1995), ACM Press, pp. 342–356.
- [8] FONSECA, R., RATNASAMY, S., ZHAO, J., EE, C.-T., CULLER, D., SHENKER, S., AND STOICA, I. Beacon-vector: Scalable point-topoint routing in wireless sensor networks. In *Proceedings of the Second* USENIX/ACM NSDI (Boston, MA, May 2005).
- [9] GANESAN, D., ESTRIN, D., AND HEIDEMANN, J. DIMENSIONS: Why do we need a new data handling architecture for sensor networks? In Proceedings of the ACM HotNets (Princeton, NJ, USA, October 2002), ACM, pp. 143–148.
- [10] GANESAN, D., GREENSTEIN, B., PERELYUBSKIY, D., ESTRIN, D., AND HEIEMANN, J. An evaluation of multi-resolution storage for sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 63–75.
- [11] GAO, J., GUIBAS, L. J., HERSHBERGER, J., AND ZHANG, L. Fractionally cascaded information in a sensor network. In IPSN'04: Proceedings of the third international symposium on Information processing in sensor networks (New York, NY, USA, 2004), ACM Press, pp. 311–319.
- [12] GREENSTEIN, B., ESTRIN, D., GOVINDAN, R., RATNASAMY, S., AND SHENKER, S. DIFS: A Distributed Index for Features in Sensor Networks. In Proceedings of First IEEE WSNA (May 2003).
- [13] HAMILTON, M., ALLEN, M., ESTRIN, D., ROTTENBERRY, J., RUNDEL, P., SRIVASTAVA, M., AND SOATTO, S. Extensible sensing system: An advanced network design for microclimate sensing, June 2003.
- [14] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed Diffusion: a scalable and robust communication paradigm for sensor networks. In Proceedings of the 6th Annual MOBICOM (2000), ACM Press, pp. 56–67
- [15] KARGER, D. R., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc.* 29th ACM STOC (May 1997), pp. 654–663.

- [16] KARP, B., AND KUNG, H. T. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th annual MOBICOM* (2000), ACM Press, pp. 243–254.
- [17] KIM, Y. J., GOVINDAN, R., KARP, B., AND SHENKER, S. Geographic routing made practical. In *Proceedings of the Second USENIX/ACM NSDI* (Boston, MA, May 2005).
- [18] KLING, R., ADLER, R., HUANG, J., HUMMEL, V., AND NACHMAN, L. The intel mote platorm: A bluetooth based sensor network for industrial monitoring applications.
- [19] KUHN, F., WATTENHOFER, R., ZHANG, Y., AND ZOLLINGER, A. Geometric ad-hoc routing: Of theory and practice. In 22nd ACM PODC (2003).
- [20] LEONG, B., LISKOV, B., AND MORRIS, R. Geographic routing without planarization. In *Proceedings of the Third USENIX/ACM NSDI* (San Jose, CA, May 2006).
- [21] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. TOSSIM: accurate and scalable simulation of entire tinyos applications. In SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems (New York, NY, USA, 2003), ACM Press, pp. 126–137.
- [22] LI, X., KIM, Y. J., GOVINDAN, R., AND HONG, W. Multi-dimensional range queries in sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 63–75.
- [23] MADDEN, S. The Design and Evaluation of a Query Processing Architecture for Sensor Networks. PhD thesis, UC Berkeley, 2003.
- [24] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TAG: a tiny aggregation service for ad hoc sensor networks. In OSDI (2002).
- [25] MAINWARING, A., POLASTRE, J., SZEWCZYK, R., CULLER, D., AND ANDERSON, J. Wireless sensor networks for habitat monitoring. In Proceedings of ACM WSNA (Atlanta, GA, Sept. 2002).
- [26] NATH, B., AND NICULESCU, D. Routing on a curve. SIGCOMM Comput. Commun. Rev. 33, 1 (2003), 137–142.
- [27] NEWSOME, J., AND SONG, D. GEM: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In Proceedings of the First SenSys (2003), ACM Press, pp. 76–88.
- [28] RAO, A., RATNASAMY, S., PAPADIMITRIOU, C., SHENKER, S., AND STOICA, I. Geographic routing without location information. In Proceedings of the 9th Annual MOBICOM (2003), ACM Press, pp. 96–108.
- [29] RATNASAMY, S., KARP, B., SHENKER, S., ESTRIN, D., GOVINDAN, R., YIN, L., AND YU, F. Data-centric storage in sensornets with GHT, a geographic hash table. *Mob. Netw. Appl.* 8, 4 (2003), 427–442.
- [30] SHENKER, S., RATNASAMY, S., KARP, B., GOVINDAN, R., AND ESTRIN, D. Data-centric storage in sensornets. SIGCOMM Comput. Commun. Rev. 33, 1 (2003), 137–142.
- [31] SZEWCZYK, R., POLASTRE, J., MAINWARING, A., ANDERSON, J., AND CULLER, D. An analysis of a large scale habitat monitoring application. In Proceedings of the Second SenSys (2004), ACM Press.
- [32] TSUCHIYA, P. F. The Landmark Hierarchy: a new hierarchy for routing in very large networks. In ACM SIGCOMM (1988), ACM Press, pp. 35–42.
- [33] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 14–27.
- [34] XU, N., RANGWALA, K., CHINTALAPUDI, D., GANESAN, D., BROAD, A., GOVINDAN, R., AND ESTRIN, D. A wireless sensor network for structural monitoring. In *Proceedings of the Second SenSys* (2004), ACM Press.
- [35] ZHAO, J., AND GOVINDAN, R. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First Sen-Sys* (2003), ACM Press, pp. 1–13.

Geographic Routing without Planarization

Ben Leong, Barbara Liskov, and Robert Morris

MIT Computer Science and Artificial Intelligence Laboratory

{benleong, liskov, rtm}@csail.mit.edu

Abstract

We present a new geographic routing algorithm, *Greedy Distributed Spanning Tree Routing (GDSTR)*, that finds shorter routes and generates less maintenance traffic than previous algorithms. While geographic routing potentially scales well, it faces the problem of what to do at local dead ends where greedy forwarding fails. Existing geographic routing algorithms handle dead ends by planarizing the node connectivity graph and then using the right-hand rule to route around the resulting faces.

GDSTR handles this situation differently by switching instead to routing on a spanning tree until it reaches a point where greedy forwarding can again make progress. In order to choose a direction on the tree that is most likely to make progress towards the destination, each GDSTR node maintains a summary of the area covered by the subtree below each of its tree neighbors. While GDSTR requires only one tree for correctness, it uses two for robustness and to give it an additional forwarding choice.

Our simulations show that GDSTR finds shorter routes than geographic face routing algorithms: GDSTR's stretch is up to 20% less than the best existing algorithm in situations where dead ends are common. In addition, we show that GDSTR requires an order of magnitude less bandwidth to maintain its trees than CLDP, the only distributed planarization algorithm that is known to work with practical radio networks.

1 Introduction

Geographic routing algorithms [2, 12, 16, 17, 20] are an attractive alternative to traditional ad hoc routing algorithms [8, 23, 24] for wireless networks, because they scale better: the routing state maintained per node is dependent only on local network density and not on network size [10, 13]. Recently, geographic routing algorithms have also been proposed as a routing primitive for

data-centric applications [21, 28]. Even when physical locations are not available, geographic routing can still be applied using virtual coordinates [22, 26].

All previously proposed geographic routing algorithms are based on *face routing* [15], which guarantees packet delivery by routing on a planar subgraph of the network. It turns out that distributed planarization is difficult for real wireless networks [11] and the problem was only solved recently by Kim et al. with the Cross-Link Detection Protocol (CLDP) [13]. However, CLDP is complex and somewhat costly, while face routing requires the handling of many subtle corner cases [14]. While practical distributed planarization is now a solved problem, the high maintenance costs and complexities associated with the deployment of face routing algorithms (with CLDP) make it worthwhile to consider an alternative approach to face routing.

We have developed a new geographic routing algorithm, the *Greedy Distributed Spanning Tree Routing (GDSTR)* algorithm, that does not require planarization. GDSTR is better than existing geographic face routing algorithms in the following respects:

- It requires significantly less maintenance bandwidth than CLDP;
- It achieves lower path and hop stretch than existing geographic face routing algorithms; and
- It is simpler and easier to understand and implement.

Like existing geographic routing algorithms, we assume that nodes have assigned coordinates and that links are bi-directional. Unlike some previous work, we do not require radio ranges to be uniform and to cover unit disks [2, 12].

Geographic routing algorithms forward packets greedily whenever possible, by routing through a directly connected neighbor in the direction of the ultimate destination. When there is no such neighbor, face routing algorithms avoid the obstacle by forwarding around the faces of a planar subgraph of the network graph. GDSTR, in contrast, switches to forwarding along the edges of a spanning tree.

A common technique for achieving scalability in traditional networking is the aggregation of information about the address space. A key insight of our work is that GDSTR can apply the same principle to help it route along its spanning tree by aggregating the locations covered by subtrees using convex hulls. We call a tree annotated with convex hulls a *hull tree*. GDSTR uses the convex hulls to decide which direction in the tree is most likely to make progress towards a given geographic destination.

GDSTR requires only one hull tree for correctness. However, we use a second tree because doing so provides better robustness in the event of node failures and an additional routing choice.

A simulation evaluation shows that GDSTR achieves a peak improvement of about 20% in terms of path and hop stretch over the best available geographic face routing algorithm in situations where dead ends are common, and that GDSTR performance is consistently good over a wide range of network densities and sizes.

Simulation also shows that GDSTR generates significantly less maintenance traffic than CLDP. GDSTR sends two orders of magnitude fewer messages to build its trees initially than what CLDP sends to construct a planar subgraph, and GDSTR's communication when maintaining existing trees is one order of magnitude less than CLDP.

The remainder of this paper is organized as follows: in Section 2, we provide a review of existing and related work. In Section 3, we describe the maintenance of hull trees and the GDSTR routing algorithm in detail, and explain why hull trees work in practice. We describe our simulation methodology in Section 4 and present our simulation results in Sections 5 and 6. Finally, we conclude in Section 7.

2 Related Work

The early proposals for geographic routing were simple greedy forwarding schemes that did not guarantee packet delivery in a connected network [3, 7, 31]. Packets are simply dropped when greedy forwarding causes them to end up at a local minimum.

The first geographic (or geometric) routing algorithm to provide guaranteed delivery was *face routing* [15] (originally called Compass Routing II). Several practical algorithms that are variations of face routing have since been developed, including GFG [2], GPSR [12] and the GOAFR+ family of algorithms [16, 17]. The

latest addition to the family is GPVFR, which improves routing efficiency by exploiting local face information [20]. While GOAFR+ is asymptotically optimal and bounds worst-case performance with an expanding ellipse search, GPVFR generally achieves the best average case stretch performance among existing geographic face routing algorithms.

The planarization algorithms that were initially available [5, 33] relied on the assumption that the underlying network is a Unit Disk Graph (UDG) for correctness and were unusable in practical networks. A major breakthrough was made by Kim et al. in developing the Cross-Link Detection Protocol (CLDP) [13], which produces a subgraph on which face-routing-based algorithms are guaranteed to work correctly. Their key insight is that starting from a connected graph, nodes can independently probe each of their links using a righthand rule to determine if the link crosses some other link in the network. CLDP uses a two-phase locking protocol to ensure that no more than one link is removed at any time from any given face; in this way it guarantees that the removal of a crossed link will not disconnect the network. While CLDP is able to planarize an arbitrary graph, every single link in the network has to be probed multiple times and it has a high cost.

There are previous routing algorithms for ad hoc networks that also use spanning trees, though none of them leverages location information like GDSTR. Radhakrishnan et al. first proposed the use of a set of distributed spanning trees for routing in ad hoc wireless networks [25]. Their algorithm constructs the spanning trees in an ad hoc manner and messages are delivered using a flooding-based algorithm.

Newsome and Song proposed an approach for routing in sensor networks called GEM, which embeds a labeled graph in the network topology [22]. They proposed Virtual Polar Coordinate Routing (VPCR), which routes packets on an embedded ringed tree graph. VPCR was evaluated in a regime where average node degree is about 15 and was found to achieve a stretch (which the authors refer to as *dilation*) of about 1.2. This does not compare favorably with geographic routing algorithms, since in the same regime, geographic routing algorithms are able to achieve unit stretch almost all the time. However, considering that VPCR does not require nodes to have access to location information because it assigns its own virtual polar coordinates, the achieved performance is reasonably good.

Beacon Vector Routing (BVR) [4] and HopID [35] are routing algorithms that employ a set of landmark nodes (beacons). Coordinates are assigned to nodes based on their hop count distances to the beacons. Routing is done by minimizing a distance function to these coordinates. When a packet is trapped at a local minimum, they resort

to scoped flooding. The major drawback of this approach is that it requires a large number of beacons (about 40) to achieve routing performance comparable to geographic routing algorithms. It is also somewhat cumbersome to have to specify a destination with a large set of distance vectors, and it may be costly to keep updating a node's coordinates when distance vectors change over time under network churn.

A common application of the spanning tree in the wired domain is the Ethernet spanning tree. The Ethernet spanning tree is not efficient for large networks because packets often have to be routed through the root of the tree. GDSTR does not suffer from the same problem, for several reasons. First it usually forwards packets greedily; the spanning tree is used only to route around voids and GDSTR reverts to greedy forwarding as soon as it is safe to do so. Second, the location information in the tree allows it to route efficiently. Finally, the location information also allows it to avoid routing through the root.

3 Greedy Distributed Spanning Tree Routing (GDSTR)

In this section, we describe our approach. We describe hull trees, explain how routing works, discuss why hull trees work well, and describe how hull trees are built and maintained.

3.1 Hull Trees

It is well-known that, given a spanning tree that contains all n nodes in a network, we can successfully deliver packets to any node on the network by traversing the tree in a manner analogous to a depth-first search as shown in Figure 1(a). The problem with such an approach, however, is that it is unlikely to route efficiently: the approach can guarantee that a packet will be delivered in no more than 2n-3 hops, but we need to do much better than that.

A major contribution of our work is the definition of a new kind of spanning tree, which we call *hull tree*, for use in networks where each node has an assigned coordinate. A hull tree is a spanning tree where each node has an associated *convex hull* that contains within it the locations of all its descendant nodes in the tree. Hull trees provide a way of aggregating location information and they are built by aggregating convex hull information up the tree. This information is used in routing to avoid paths that will not be productive; instead we are able to traverse a significantly reduced subtree, consisting of only the nodes with convex hulls that contain the destination point. An example of a hull tree corresponding to the spanning tree shown in Figure 1(a) is illustrated in Figure 1(b).

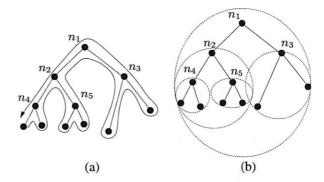


Figure 1: Examples of a spanning tree and a *hull tree*. Although convex hulls are polygons, they are represented with ellipses in Figure 1(b) for simplicity.

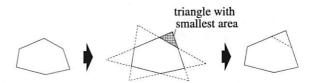


Figure 2: Procedure for reducing the size of a convex hull.

The convex hull for a set of points is the minimal convex polygon that contains all the points; it is minimal because the convex hull will be contained in any convex polygon that contains the given points. The hull is represented as a set of points (its vertices), and this set could be arbitrarily large. To ensure that the convex hulls use only O(1) storage instead of O(N) storage, where N is the network size, we can limit the number of vertices for a convex hull to a maximum of r points. To reduce a convex hull with s vertices to a smaller one with s-1 points, we can project the boundary lines to form an adjacent triangle at every face. We pick the smallest triangle in this set of s triangles and add that triangle to the hull as illustrated in Figure 2.

Limiting the number of points on the convex hulls allows us to save storage, but the resulting hulls will be larger and this increases the probability that the hulls of two siblings nodes in a tree will intersect. Intersections between convex hulls are undesirable because they introduce ambiguity in the routing process and make it less efficient. However, our experiments (described in Section 5) show that routing behavior is not affected by using as few as 5 points to represent a hull.

3.2 Overview of Routing

GDSTR forwards packets using simple greedy forwarding whenever possible. It switches to forwarding based on a hull tree only to route packets around "voids," and escape from a local minimum. It switches back to greedy forwarding as soon as it is feasible to do so.

When a packet reaches a local minimum, there are the following possibilities:

- The destination point is not contained in the convex hulls of any of the child nodes: Forward the packet to the parent node. If the packet reaches a node whose convex hull contains the destination, it will be routed down the tree from that node to reach the destination. If the packet reaches the root node and none of the convex hulls of its child nodes contain the destination, we can conclude that the packet is undeliverable.
- 2. The destination point is contained in the convex hull of at least one child node: We order the child nodes whose hulls contain the destination point using a fixed ordering based on node identifiers. The routing decision depends on (i) whether the packet was previously forwarded in greedy forwarding mode, and (ii) from which node the packet was received:
 - (a) Packet was previously in greedy mode or was received from parent node: forward packet to the first child node whose convex hull contains the destination.
 - (b) Packet was received from child node (except last child node): forward packet to the next child node whose convex hull contains the destination.
 - (c) Packet was received from last child node among those whose hulls contain the destination: forward packet to the parent node, or to the first child if node is the root node and has no parent node.

By recording the node at which we start the tree traversal in the packet, we can conclude that a packet is undeliverable when we come back to the same node. This termination condition is analogous to that used to terminate traversal of planar faces by existing geographic face routing algorithms.

Routing using hull trees is illustrated in Figure 3. Figure 3(a) shows what happens when node n_3 sends a packet to node n_5 : since n_5 is not in n_3 's convex hull, the packet will be sent first to n_1 , and from there to n_2 , since its convex hull contains the destination.

Figure 3(b) illustrates a more complex example involving an undeliverable packet. Suppose node n_4 sends a message to an unreachable destination x, and initially this packet is routed greedily to n_2 , and then to n_5 , which is a local minimum. At this point n_5 records itself in the packet and switches to routing in tree forwarding mode. The packet is forwarded on the subtree consisting of the

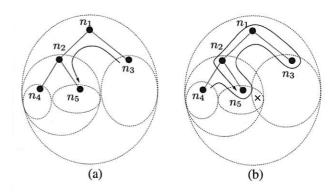


Figure 3: Routing over sample hull trees. Child nodes of n_3 , n_4 and n_5 are omitted to avoid clutter. The destination of the forwarded packet is marked with a cross (applicable to (b) only).

nodes with hulls that contain the destination (which in our example are the nodes n_1 , n_2 , n_3 and n_5). The packet is first sent to the parent node n_2 and from there to n_1 . The destination is contained in the convex hulls of both of n_1 's child nodes, but since the packet was received from n_2 , it is forwarded to n_3 . After forwarding over subtrees of n_3 (not shown on the diagram), the packet is returned to n_1 , which forwards it to n_2 , its first child whose convex hull contains the packet. n_2 forwards the packet to n_5 . At this point n_5 sees that it was the originator of the tree traversal and hence concludes that the packet is undeliverable.

3.3 Optimization for Undeliverable Packets

The example in Figure 3(b) illustrates that undeliverable packets will unfortunately always be routed to the root of a hull tree. For applications like data-centric sensor networks where the destination of packets often do not correspond to actual nodes, this situation is unacceptable. The problem arises because, during tree traversal, a node that receives a packet from its last child node does not know if there is any other convex hull in another part of the tree that contains the destination and has no choice but to forward the packet to the parent node.

To remedy this situation, we maintain additional information in the tree to allow a node to decide if the destination of a packet could possibly be in a distant branch of the tree reachable only by forwarding the packet up the tree. In addition to its convex hull, each node maintains information about the set of convex hulls \mathcal{H} that intersect with its own convex hull. We refer to these hulls as *conflict hulls*. A node stores a conflict hull for related nodes that are not descendants or ancestors, i.e., for siblings and cousins. More precisely, it stores conflict hulls for nodes with which it shares a common ancestor, where that node

is immediately below the common ancestor, and its convex hull intersects with this node's. In the example network shown in Figure 3(b), the hull of n_3 is recorded as a conflict hull by both n_2 and n_5 ; however, n_3 will record only the hull of n_2 (and not that for n_5) as a conflict hull.

With this additional information, a node that receives a packet from its last child during tree traversal will check if any of its conflict hulls contain the destination. If not, it will forward the packet to its first child instead of the parent. Effectively, the conflict hulls allow us to prune some nodes at the root of the routing subtree during tree traversal.

3.4 Using Multiple Trees

Using a single tree as the basis of routing is inherently fragile. If the root node fails, the entire tree may collapse and have to be rebuilt, and while this is happening, routing will not work well. GDSTR provides some degree of resilience to such network changes by maintaining a set of k hull trees, each of which is uniquely defined by its root node.

With multiple trees, a tree must be chosen when a packet switches from greedy forwarding to tree forwarding mode. We studied a number of heuristics [19] and found that following simple heuristic works best:

- From the set of trees that have a child node with a convex hull containing the destination node, pick any tree (at random).
- Otherwise, if none of the child nodes (in any tree) have convex hulls that contain the destination node, pick the tree with the root that is nearest to the destination.

3.5 Routing Algorithm

The following is a more precise description of GDSTR that incorporates the use of multiple trees and the set of conflict hulls \mathcal{H} . A GDSTR data packet p is tagged with the following state components:

- mode: current forwarding mode (Greedy/Tree),
- n_{min}: node that is nearest to destination,
- tree: identifier for chosen forwarding tree,
- n_{anchor}: tree traversal anchor node.

 n_{min} is the node at which a packet switches from greedy forwarding to tree traversal. It is used to determine when routing should revert to greedy forwarding. n_{anchor} is the first node encountered by a packet during tree traversal that has a convex hull containing the destination.

While n_{min} is often the same as n_{anchor} , they are occasionally not the same node. n_{min} is used by a node to determine whether is it safe for a packet to revert to greedy forwarding mode and n_{anchor} is used to determine if the packet is undeliverable during tree traversal. We use a tree building algorithm that guarantees the uniqueness of a tree given a root node. Hence, the tree identifier on a packet is the node identifier of the root node of the hull tree.

Algorithm (GDSTR). When a node v receives packet p for destination node t from a neighboring node u, do:

Check for switch to Greedy mode: If p.mode = Tree and there is at least one immediate neighbor w such that $|wt| < |(p.n_{min})t|$, then set p.mode := Greedy, $p.n_{min} := w$ and clear $p.n_{anchor}$ and p.tree if they are set. Execute step 2 or 3 according to p.mode.

- Greedy Mode: Find the node w in the set of immediate neighbors that is closest to t. If |wt| < |vt|, forward the packet to w. Otherwise, set p.mode := Tree and follow step 3.
- 3. **Tree Mode:** If p.tree is not set or if the root for p.tree has changed, follow step 4, else follow step 5.
- Choose Hull Tree: Choose one of the existing hull trees for forwarding and set p.tree to the chosen tree's identifier. Follow step 5.
- 5. Check Hull Tree: If hull tree does not contain destination node t, follow step 6; otherwise, follow step 7 instead.
- 6. Not in Hull Tree: If none of the hulls in H contains the destination node t conclude that the packet is not deliverable. Otherwise, forward p to the parent node in p.tree.
- 7. Check Anchor Node: If $p.n_{anchor}$ is set, follow step 8. Else, set $p.n_{anchor} := v$ and follow step 9 instead.
- 8. Termination Condition: Given a global ordering for node identifiers, arrange the child nodes (relative to p.tree) with convex hulls that contain the destination point in a ascending sequence according to the global ordering. Then:

- If v = p.anchor and either (i) u is the last child and v is the root node for p.tree, (ii) u is the last child and the set of conflict hulls $\mathcal H$ does not contain destination node t, or (iii) u is the parent node, conclude that packet is **not deliverable**; else
- If u is the parent node and the set of conflict hulls H does not contain destination node t, set p.n_{anchor} := v. Follow step 9.

9. Tree Traversal:

- If p.mode = Greedy, set p.mode := Tree, forward packet to the first child node;
- If packet was received from the parent node, forward packet to the first child node;
- If packet was received from a child node, forward packet to the next child node in the sequence;
- If packet was received from the last child node, forward packet to the parent node if one of the hulls in H contains the destination point; else, forward packet to the first child node.

The correctness of this algorithm follows from the fact that the geometric properties of the routing subtree will ensure that a packet will eventually visit every node that can possibly be its destination. Termination is guaranteed because $|(p.n_{min})t|$ is strictly decreasing while a packet is in greedy forwarding mode; and if a packet is not deliverable, it will eventually return to the anchor node n_{anchor} in tree traversal mode [19].

3.6 Why and How Hull Trees Work

The common wisdom about spanning trees is that they result in low total performance: routes can only use tree links, leaving the majority of links idle. However, this drawback does not apply to GDSTR for two reasons.

The first reason is that GDSTR uses tree routing only when greedy geographic forwarding encounters a dead end. Furthermore GDSTR switches back to greedy forwarding as soon as possible. Thus typically only a small fraction of hops use tree routing and few packets are routed through the root nodes of the hull trees. Since greedy forwarding yields good stretch when it works [34], GDSTR provides good overall performance.

The second reason GDSTR performs well is that it does a good job of routing around voids. GDSTR

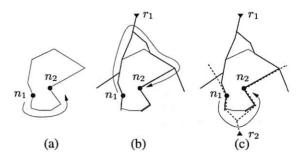


Figure 4: An example illustrating why it helps to have two trees when traversing a void.

achieves this by using hull trees that are rooted at the extremities of the network, which allows it to find routes that conform closely to the voids.

Figure 4 illustrates this point. Suppose that node n_1 in Figure 4(a) needs to route a packet around a void to n_2 . A face routing algorithm is likely to have a face exactly corresponding to the void, and must choose between routing clockwise or counter-clockwise. In this example, the optimal choice is counterclockwise. While having some local face information allows a face routing algorithm to pick the optimal direction fairly often [20], local information alone cannot guarantee that the optimal routing direction will be picked when the void is large.

Suppose GDSTR is the routing algorithm and has one hull tree rooted at r_1 . As illustrated in Figure 4(b), n_1 would be forced to route clockwise. However, with another tree rooted at r_2 , at the opposite end of the network, n_1 is presented with the other choice as well, as illustrated in Figure 4(c). This example demonstrates how two trees can effectively "approximate" a planar face.

This insight into how the hull trees work also explains the expected performance trade off between GDSTR and face routing algorithms. Face routing techniques are able to traverse voids in a wireless routing topology relatively efficiently. The key issue is that they often do not have sufficient information to choose the optimal forwarding direction and it can be very costly when they make a bad choice. It turns out that with GDSTR where there are at least two extremally-rooted trees, the simple heuristic of choosing the tree with a root that is nearest to the destination often allows GDSTR to choose the more favorable forwarding direction around a void (because it effectively has global information).

For sparse networks with large voids, GDSTR is thus able to perform significantly better than geographic face routing algorithms. For dense networks, the voids tend to be small and it generally does not matter which forwarding direction is picked. Because hull trees are not able to approximate voids quite as well as planar faces, face routing is expected to achieve slightly better stretch than GDSTR in such cases.

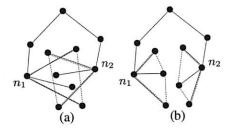


Figure 5: Examples of "bad" and "good" trees.

3.7 Building and Maintaining Hull Trees

This section explains how we build and maintain hull trees.

Choosing Root Nodes. Since our goal is to use hull trees to "approximate" voids, we want the roots of the k hull trees as far apart as possible. To achieve this, we choose k rays in different directions, rooted at the origin, and the roots are the nodes whose projections onto these rays are farthest from the origin. If there are multiple extremal nodes, we break ties by imposing a global ordering on the node identifiers. For example, to maintain two trees, we can use one rooted at the node with the maximal x coordinate and the other rooted at the node with the minimal x coordinate.

Each node broadcasts a *keepalive* message periodically to inform its neighbors of its location. The node includes in each message its view of the root of each tree and its distance in both hop count and path distance from each root. Through these exchanges, all the nodes will eventually come to a consensus as to which nodes should be roots; each node will also know both its hop count and total path distance from the root.

Spanning Tree Algorithm. GDSTR will work correctly with any distributed spanning tree. However, routing performance will be best if there is minimal overlap among the convex hulls of different tree branches. Intuitively, we want trees that are geographically "compact." Figure 5 illustrates this idea. While the nodes in both Figures 5(a) and 5(b) are the same, the tree configuration in Figure 5(a) creates an undesirable intersection in the hulls for nodes n_1 and n_2 . From these examples, it is clear that we want to build trees that cluster nearby nodes in the same subtree. In addition, we want to be able to route from the root of the tree to all the nodes in the network in a small number of hops, since this will likely reduce the routing hop count.

After evaluating a number of spanning tree algorithms, we found that the *minimal-path tree* seems to work best. The *minimal-path tree* is constructed by having each node choose the neighbor with the minimal path distance to the root as its parent (and updating its path distance accordingly). The details and simulation results for the other spanning tree algorithms are contained in [19].

Building Hull Trees. Once the tree has been formed, each node broadcasts its chosen parent node as well as its convex hull. To compute its convex hull, a node determines the minimal convex hull that contains the union of the convex hulls of its children in that tree. The convex hull for a set of points can be computed in $O(n \log n)$ operations using the Graham's Scan algorithm [6].

Once the root acquires hulls from all its children, the final step is for each node is to determine the set of conflict hulls \mathcal{H} and add this information to its *keepalive* messages. Information about the conflict hulls is propagated down the tree starting at the root; each node in turn informs its children about intersections between their hulls and other known hulls. Once the information about the conflict hulls has propagated down to the leaves of the tree, the tree is fully built and consistent. This algorithm (like other tree building algorithms) takes at most 3D rounds of message exchanges to complete, where D is the diameter of the network graph.

Maintenance & Repair. We use the same algorithm to repair a tree when nodes fail. If the mean intermessage interval is T seconds, even in the worst case where the root of a tree fails, a hull tree can be restored within 3TD seconds. To speed up tree repair and recovery, we can trigger immediate transmissions in place of regular messages when a node failure is detected.

A node concludes that a neighbor has failed when it does not hear from it after a pre-determined multiple of the *keepalive* message interval. If the failed node is a child, a node will reduce and update its convex hull; if the failed node is a parent, a node will choose a new parent. In either case, it sends the new information in its next *keepalive* message. When the (new or old) parent hears about the changes, it will update its state accordingly.

Hence, it is straightforward to update the routing state when anything changes in the system. When a node hears the *keepalive* message from a neighbor, it updates its own state and the information that it broadcasts in its subsequent *keepalive* message. If nothing changes, a node does not need to update anything.

In fact, a node only has to broadcast its hull tree information when there are changes to the state of its hull trees. If nothing changes after the same hull tree information has been sent for several rounds, subsequent *keepalive* messages will contain only the node's identifier and location. When there is a change in its hull tree information, a node resumes broadcasting its hull tree information for another few rounds.

Assumptions. The spanning tree algorithm makes few assumptions about radio behavior. The only requirement is that nodes must agree about whether they are neighbors. GDSTR is also robust to location errors [29], because if a node has a wrong location, the hulls in its part of the hull trees will grow to include the node's wrong

location. When greedy routing to that node hits a dead end, GDSTR's tree traversal will eventually route to the tree branch that includes the node because of the large hull.

4 Simulation Setup

We evaluated the performance of GDSTR with simulations and this section describes our simulation setup. The simulations are performed using our own high-level event-driven simulator [18]. Sections 5 and 6 will present the simulation results.

For our simulations, we use a simple radio model: all nodes have unit radio range; two nodes can communicate if and only if they are within radio range of each other and if their line-of-sight does not intersect an obstacle. The simulator supports linear, polygonal and circular obstacles. Wireless losses are not simulated since our goal is to compare the basic algorithmic behavior of GDSTR to other geographic routing algorithms.

As discussed in Section 3, the underlying radio model does not matter for GDSTR. While the simulator is able to support non-uniform radio ranges, we consider only topologies with uniform unit radios since uni-directional links are not used by GDSTR and topologies with non-uniform radio ranges can be replicated by adding obstacles. Even under this assumption, we are able to generate a diverse range of topologies, which we believe is adequate for the purposes of comparing GDSTR to existing geographic face routing algorithms.

Effect of Network Density: To understand the effects of network density on routing performance and maintenance costs, we generated networks with 25 to 500 nodes randomly scattered over a 10×10 unit square. This process generated networks with average node degrees between 0.7 to 14.4. For each density, we generated 200 networks, and then routed 20,000 packets using each algorithm between randomly chosen pairs of source and destination nodes. The performance measurements presented are the average over the 200 times 20,000 data points. We also used these topologies to evaluate the effects of parameters like the number of hull trees and the value of r, the maximum size for the convex hulls.

A density of 500 nodes in 100 square units is high enough that greedy forwarding almost always succeeds, and neither GDSTR nor face routing is needed. For this reason we did not explore higher densities.

Effect of Obstacles and Network Size: To evaluate the scaling of maintenance costs and performance and the effect of obstacles, we generated a range of networks with constant node density from 50 to 5,000 nodes in size. The networks were generated for each size by scattering nodes randomly over an $x \times x$ unit square, where x was scaled by a factor of \sqrt{n} for each network size

n. In addition, we also added a number of cross-shaped obstacles (0.25 units across) proportional to the size of the area over which nodes are scattered. This procedure sometimes generated networks that were not connected. We discarded such networks and repeated the above procedure until we had 200 connected networks for each network size. We found that it is difficult in practice to generate random connected networks for graphs with a density of obstacles above a given threshold.

We investigated scaling effects on networks by varying the density of obstacles. The average node degrees of the studied networks ranged from 7 to 11. We are interested in networks of these densities because they are close to the critical region in which existing geographic routing algorithms are known to perform poorly [17].

Comparisons. We compare the routing performance of GDSTR to GPSR [12], GOAFR+ [16] and GPVFR [20]. We evaluate the geographic face routing algorithms (GPSR, GOAFR+ and GPVFR) with CLDP planarization, rather than *Gabriel Graph* (GG) [5] or *Relative Neighborhood Graph* (RNG) [33] planarization, since CLDP is currently the only algorithm that is known to work for practical networks. In any case, the networks with obstacles are not unit-disk graphs (UDGs) and hence GG and RNG would not planarize them correctly.

Our implementations of these routing techniques are based on the algorithms described in [12], [16] and [20] respectively. The configuration parameters for GOAFR+ are $\rho_0=1.4$, $\rho=\sqrt{2}$ and $\sigma=\frac{1}{100}$ as suggested in [16] and for GPVFR, we limited the length of the propagated path vectors to 3. Unless stated otherwise, we used two hull trees for all experiments with GDSTR. Our implementation of CLDP follows the description in [13].

5 Routing Performance

In this section, we compare the performance of GDSTR routing to existing geographic face routing algorithms.

We measured routing performance with respect to two metrics: (i) path stretch, and (ii) hop stretch. Path stretch is the ratio of the total path length to the shortest path (in Euclidean distance) between two nodes; hop stretch is the ratio of the number of hops on the route between two nodes to the number of hops in the shortest path (in terms of hops). It turns out that results for these two metrics are similar, so we present only the plots for hop stretch below.

5.1 Routing Performance

Figure 6 shows the hop stretch for deliverable packets for GDSTR, GPSR, GOAFR+ and GPVFR over a range of average node degrees.

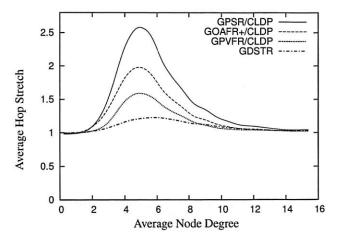


Figure 6: Plot comparing the hop stretch for GDSTR to that for GPSR, GOAFR+ and GPVFR under CLDP planarization for unit radio networks in a 10×10 unit square.

GDSTR has the best performance for most of the range: that is, GDSTR routes packets along shorter paths than the other algorithms, and is thus likely to deliver packets faster and with less consumption of radio resources. The only exception is that GPVFR's stretch is a few percent less than GDSTR for node degrees higher than 9.

The differences in the performance of the various algorithms are most pronounced in the critical region of node degrees between 4 and 6. The reason is that networks in this region tend to have large outer perimeters and the voids that are generated are often concave. Packets tend to end up in a local minima fairly often for these topologies and the routing algorithm has to resort to forwarding the packet along a face (for the face routing algorithms) or along a tree (for GDSTR).

GPSR performs the worst because it uses a deterministic *right hand rule* when forwarding a packet along a face. It turns out that topologies in the critical region typically present nodes that need to switch to face traversal with one good forwarding direction and one terrible alternative. By choosing the same direction consistently, GPSR gets it wrong about half the time.

GOAFR+ is better than GPSR because it uses an expanding ellipse to bound the search radius. GOAFR+ picks a random forwarding direction to start with, but instead of forwarding continuously along a face, GOAFR+ keeps track of how far it has gone along the face and if a packet seems to have wandered far enough along a face and not made any apparent progress toward the destination, GOAFR will make the packet backtrack. By expanding the area of the search incrementally, GOAFR ensures that the length of the final path traversed is no longer than a constant multiple of the optimal path.

GPVFR tries to pick the optimal forwarding direction when it switches from greedy forwarding to face traversal. It does so by maintaining several hops worth of information about its adjacent planar faces. It turns out that in practice, by maintaining information about nodes that are up to 4 hops away along the planar faces, GPVFR will often make the correct decision when the network density is low. When the network density is relatively high (above an average node degree of 9), CLDP produces planar faces that are relatively small (usually with fewer than seven points). Thus, under such circumstances, GPVFR has enough information to guarantee that it chooses the correct forwarding direction almost all the time, which explains why it performs better than GDSTR for node degrees higher than 9.

When two forwarding directions are available, GDSTR's tree-choosing heuristic of picking the tree with a root that is closest to the destination allows us to choose a good forwarding direction around a void most of the time. However, we believe that a more significant reason that explains why GDSTR outperforms the other algorithms in the critical region is that the convex hulls contain sufficient information to allow GDSTR to prune away many bad routing choices and route on a much reduced subtree that is often significantly smaller than large voids or the perimeter of the network (which often have one hundred or more nodes).

5.2 How Many Trees are Useful?

GDSTR can maintain multiple hull-trees. Figure 7 shows the effect of increasing the number of trees on the average hop stretch. Routing performance improves quite significantly when we increase the number of hull trees from one to two (achieving a peak improvement of approximately 10% in path and hop stretch); routing performance continues to improve with more trees but beyond two trees, the improvement is marginal. This is not surprising since two extremally-rooted trees are sufficient to approximate voids relatively well.

5.3 Effect of Convex Hull Representation

In Sections 5.1 and 5.2, we did not limit r, the maximum size for the convex hulls.

When we repeated the measurements for routing stretch for different values of r, we found that surprisingly, the value of r has a negligible effect on both path and hop stretch, i.e., the stretch for r=5 was virtually indistinguishable from stretch when r is unlimited. We found that the reason for this is that although the hulls are bigger when r is limited and there are more intersections between the convex hulls of sibling nodes, intersections do not necessarily degrade routing performance as long

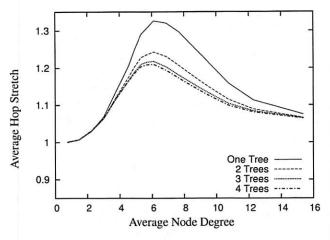


Figure 7: Hop stretch performance for GDSTR with different numbers of hull trees.

as they are not particularly large or if they occur close to the leaves of a tree. In fact, intersections that do not contain any nodes do not affect routing performance.

It seems that even when r is reduced and the size of the hulls is increased, it is still relatively rare for nodes to fall into the intersections of hulls. Furthermore, intersections only matter when a packet is not forwarded in greedy mode. Since GDSTR forwards packets in greedy mode more than 75% of the time in our experimental setup and only occasionally switches to tree forwarding mode, it is not completely surprising that r does not seem to affect the aggregate routing performance.

5.4 Scaling Up

To understand how the routing performance of GDSTR scales with the size of the networks and also its performance on topologies that are not unit disk graphs (UDGs), we evaluated the routing performance of GDSTR on sets of networks with cross-shaped obstacles for sizes ranging from 50 to 5,000 nodes, while holding both node and obstacle densities constant. Without obstacles, the average node degrees of these networks would be 10; with the addition of obstacles, the average node degrees are reduced accordingly.

The hop stretch for networks with average node degrees approximately 6 and 7 are shown in Figures 8 and 9 respectively. These results are similar to that for random unit disk graphs with average node degrees 6 and 10 respectively [19].

Our results demonstrate that for sparse networks, the routing performance of GDSTR is consistently better than that for existing face routing algorithms, while for denser and larger networks, existing face routing algorithms can sometimes achieve slightly lower stretch. As mentioned, the reason is that extremally-rooted trees do

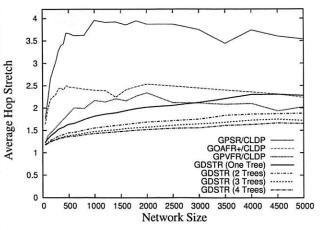


Figure 8: Plot of hop stretch for non-UDG networks (with obstacles) of average node degree 6.

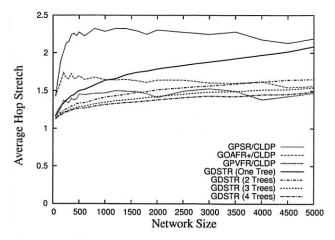


Figure 9: Plot of hop stretch for non-UDG networks (with obstacles) of average node degree 7.

not approximate voids quite as well when there are a large number of hops between the leaf nodes and the root.

6 Costs

In this section we present experimental results for the costs of GDSTR. Our main concern is with bandwidth since it is likely to be a limiting factor in radio networks. However we begin by discussing the storage costs of our system, since storage concerns were once a primary motivation for geographic routing algorithms.

6.1 Storage Costs

Figure 10 shows the average and maximal storage required by any nodes over the range of densities investigated. We assume that a set of coordinates and a node identifier are 8 and 12 bytes in size respectively. We

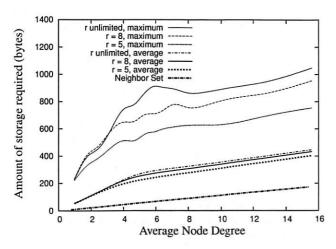


Figure 10: Amount of routing state stored at each node for various values of r for GDSTR with two hull trees.

can see from the figure that the maximum is about 1,000 bytes. This amount of storage is hardly a concern for modern sensor devices like the Mica2 [32], which has 128K of program memory and 512K of flash RAM.

The figure shows the storage requirements when GDSTR uses two hull trees. In general, GDSTR with two hull trees requires more than twice as much storage on average as existing face routing algorithms at low network densities. However, as the network density increases, the storage requirement of the neighbor set becomes comparable to the storage requirement for the hull trees.

The figure also shows the effect of limiting the size of the convex hulls, r, on storage. These results show that by limiting r, there is negligible effect on the average storage requirement. When r=5, we can reduce the maximum storage required by up to 30% at low network densities. Since the associated storage costs are small, we find that there is no compelling reason for us to limit the size of the convex hulls in practice.

6.2 Bandwidth Costs

In the following experiments that measure the costs of stabilization and maintenance, we compare the costs of GDSTR with the cost of building and maintaining a planar graph with CLDP. The reason is that the other associated costs of existing geographic face routing algorithms [12, 16, 20] are small relative to the cost of CLDP. The costs for GPSR [12] and GOAFR+ [16] are negligible; GPVFR [20] does impose some maintenance cost on the network to maintain its face information, but the cost is also small relative to CLDP.

We quantify the bandwidth costs for each algorithm in terms of the number of messages sent or forwarded

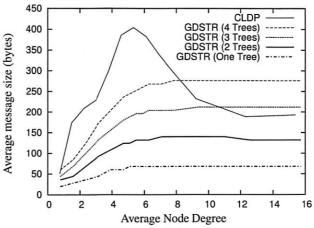


Figure 11: Comparing the sizes of CLDP probes and GDSTR broadcast messages.

by nodes during stabilization and repair. For GDSTR, we count the number of *keepalive* messages that contain new hull information. For CLDP, we count the probe messages.

The average size of these messages is shown in Figure 11. As shown, the relative sizes of the CLDP probes and GDSTR broadcast messages are comparable. CLDP probes are largest in the critical region (average node degree 4 to 8) because the probes contain the points on the faces and these networks tend to have the largest perimeters.

Startup Costs. To investigate the startup costs for a network, we start all the nodes in the network at approximately the same time and measure the average number of messages sent by each node before the network converges.

CLDP involves a locking mechanism, so a configuration involving binary backoff will likely be able to optimize its startup performance. We do not know the optimal parameters, so we used the following simple probe model: all nodes have the same probing period with a 20% jitter (to avoid synchronization), and at the start of each period, a node probes all the links that require probing. If a reply is received, it is acted on immediately. If a probe message is dropped because it encounters a locked edge, the node will resend the probe during the next probe interval. A node is deemed to have converged when all its links are marked either dormant or nonroutable and it does not have to initiate any more probes during the next probing interval. Similarly, a GDSTR node is deemed to have converged when it no longer needs to broadcast hull information in its keepalive messages.

In Figure 12, we plot the average number of messages that each node in the network would have sent or

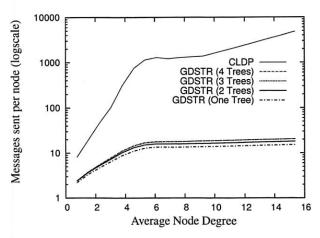


Figure 12: Packets sent or forwarded per node for stabilization.

forwarded before the network stabilizes, with all nodes starting up without any state. As shown, CLDP sends about two orders of magnitude more messages than GDSTR before the network stabilizes. For node degrees between 6 and 14, each CLDP node will send about 1,500 messages; for GDSTR, the corresponding number is slightly more than 10 messages.

There are two main reasons why CLDP imposes such a high overhead:

- Many links to probe. When the network just starts up, all links are routable and the effective forwarding graph is highly non-planar. Every edge is probed at least twice, once by each node on either side. Also, the final probes that allow a node to mark a link as dormant must fully traverse a face, so the number of messages required to probe a face grows quadratically with the size of the face;
- Locking mechanism causes packets to be dropped. Every link removal operation consists of a prepare step and a commit step. In between the two, an edge is locked and packets that arrived at it are dropped.

We see in Figure 12 that the startup costs for CLDP increases rapidly until node degree 6 and starts to taper off thereafter. This is because below node degree 6, the experimental topologies usually consist of several small disjoint networks. As the node degree increases, the networks become larger and more tree-like, and they tend to have larger perimeters that are costly to probe. After a critical density of about node degree 6, the networks become more connected, and their perimeters are somewhat more convex. The probing costs do not increase much at this stage with increasing density because the network perimeters either stay relatively constant or may even shrink slightly. The probing costs for CLDP are also

proportional to the number of edges in the network graph however, so when the network density increases beyond node degree 8, the increase in the number of edges (links) becomes the dominating factor and we again see an increase in the CLDP probing cost.

Figure 12 also shows that the number of messages required per node by GDSTR plateaus at node degree 6 and increases only slightly thereafter. The reason for this is that the number of update messages that GDSTR requires is a function of the network diameter D. It turns out that since the nodes have unit radio range and are all contained within a 10×10 unit square, D is somewhat constant for densities higher than node degree 6.

Incremental Costs. To quantify the bandwidth required to update routing state when a new node joins and to repair routing state after a node fails, we measure the costs of adding and removing a single node from a stable network as follows: after a network has stabilized, we remove one node and count the number of messages sent per node. After the network has stabilized once again, we add the removed node back to the network and take the same measurement. We repeat this process on 20 randomly chosen nodes for each network and average the results to obtain the average cost per node change in each network.

In Figure 13, we plot the number of messages that are sent per node in order for the system to converge after one node join or departure. The peak for CLDP is about 200 messages per node at a node degree of 6. When a node joins the network for CLDP, new links are created between it and all its immediate neighbors and these new links are probed independently by the various nodes; when a node fails, its adjoining neighbors will probe all the adjacent links that are marked non-routable, in case there is the need to revive a non-routable link to restore connectivity. We see that the costs for node joins and departures are comparable, except for high network densities. The likely reason for this is that at high densities, node failures are significantly more costly than node joins because more links are re-probed for node failures and the number of such links is proportional to the node degree.

The join costs for GDSTR are uniformly low at approximately 3 messages per node; the repair costs after a node failure are highest in the region with node degree between 2 and 6 and falls gradually with increasing node density. The latter is because the likelihood of failure for an intermediate node is much higher at lower node densities (with a maximum of approximately 15 messages) and for high node densities, node failures are more likely to occur at the leaf nodes. Note that these figures are averaged over only the nodes that are affected by a node join or departure.

The bandwidth costs for updating a planar graph with

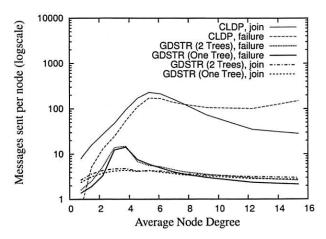


Figure 13: Packets sent or forwarded per node when a new node joins or when an existing node fails.

CLDP incrementally are significantly lower than that for *en masse* stabilization at startup. In fact they can be interpreted also as the cost to stabilize for CLDP when the network grows one node at a time.

Discussion. We chose to evaluate startup and maintenance in terms of message count rather than convergence time because the system parameters for both CLDP and GDSTR can be tuned to achieve faster convergence. For example, the probing rate for CLDP can be increased and for GDSTR, nodes can broadcast update messages as and when there are changes in its hull trees instead of waiting to piggyback the information on *keepalive* messages. The total amount of information to be transmitted to bring the routing information to a consistent state is however the same in all cases. In fact, we can work out the fundamental limit on convergence time by dividing the volume of messages to be transmitted by the maximal achievable bandwidth of the radios.

6.3 Scalability

In this section, we summarize what happens to cost as we scale up the network size to 5,000 nodes for the networks with cross-shaped obstacles.

Storage Costs. The average storage required per node is somewhat independent of network size and is about 300 bytes over the entire range of network topologies that we investigated; the maximal storage requirement increases steadily with network size, but it does not exceed 1,300 bytes even when the network size is scaled up to 5,000 nodes.

Bandwidth Costs. For large networks, the initial startup costs where all nodes start from a fresh state is not important, since large networks will have to be turned on incrementally. In Figure 14, we plot the average number of messages that are sent per node for CLDP and GDSTR for increasing network size for networks with mean node

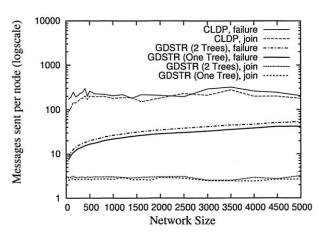


Figure 14: Packets sent or forwarded per node when a new node joins or when an existing node fails for networks with cross-shaped obstacles and mean node degree 7. See Figure 9 for corresponding routing performance.

degree 7 and a constant obstacle density.

The average number of messages sent per node for CLDP seems independent of the network size. For GDSTR, we see that messages sent per node for each incremental node join and network repair increase very gradually with network size. As before, node joins for GDSTR are relatively cheap, while repairs after network departures are slightly more costly. Individual node joins and network repairs only affect a fraction of the nodes in the network (typically less than 20% and decreasing with increasing network size).

7 Conclusion

The key insight of our work is that for geographic routing, it is no less efficient to use two hull trees instead of a planar graph as the backup routing topology when greedy forwarding fails, and it is significantly easier to build and maintain hull trees than a planar graph. Our simulations have demonstrated that GDSTR requires an order of magnitude less maintenance bandwidth than CLDP, while achieving lower path and hop stretch than existing geographic face routing algorithms.

GDSTR is immediately applicable to a large class of stationary wireless networks, e.g. roofnets [1, 30] and sensornets [9, 27]. While we have not explicitly evaluated the performance of GDSTR for mobile networks, our simulations show that GDSTR requires only a small number of packets to set up and repair its hull trees. This suggests that it is quite plausible that GDSTR will work well in a mobile setting with some tuning and optimization. It remains as future work to implement and evaluate GDSTR in a practical mobile environment.

While GDSTR is currently implemented over twodimensional Cartesian coordinates, it is generalizable to coordinates in higher dimensional spaces, since convex hulls are generalizable to higher dimensions. An open question is whether GDSTR can achieve better routing stretch in higher dimensional space.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and Dan Rubenstein for shepherding this paper. We also thank Eric Demaine, Young-Jin Kim, George Lee, Ji Li, Sayan Mitra, David Schultz, and Jijon Sit for their helpful comments on the early drafts of this paper.

References

- D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Linklevel measurements from an 802.11b mesh network. In *Proceed-ings of ACM SIGCOMM Conference* 2004, August 2004.
- [2] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. Wireless Networks, 7(6):609–616, 2001.
- [3] G. G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. Technical Report ISI/RR-87-180, ISI, March 1987.
- [4] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable pointto-point routing in wireless sensornets. In *Proceedings of NSDI* 2005, pages 329–342, May 2005.
- [5] K. Gabriel and R. Sokal. A new statistical approach to geographic variation analysis. Systematic Zoology, 18:259–278, 1969.
- [6] R. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Letters*, 1:132–133, 1972.
- [7] T. Hou and V. Li. Transmission range control in multihop packet radio networks. *IEEE Transactions on Communications*, 34(1):38–44, 1986.
- [8] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353, 1996.
- [9] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of Mobicom* '99, pages 271–278, August 1999.
- [10] B. Karp. Geographic Routing for Wireless Networks. PhD thesis, 2000.
- [11] B. Karp. Challenges in geographic routing: Sparse networks, obstacles, and traffic provisioning, May 2001.
- [12] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proceedings of Mobicom 2000*, pages 243–254, August 2000.
- [13] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *Proceedings of NSDI 2005*, pages 217– 230, May 2005.
- [14] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. On the pitfalls of geographic face routing. In *Proceedings of DIAL-M-POMC* 2005, September 2005.
- [15] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Confer*ence on Computational Geometry, pages 51–54, August 1999.

- [16] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of PODC* 2003, pages 63–72, July 2003.
- [17] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-Case Optimal and Average-Case Efficient Geometric Ad-Hoc Routing. In Proceedings of MobiHoc 2003, pages 267–278, June 2003.
- [18] B. Leong. Geographic routing network simulator, 2004. http://web.mit.edu/benleong/www/netsim.
- [19] B. Leong. New Techniques for Geographic Routing. PhD thesis, 2006
- [20] B. Leong, S. Mitra, and B. Liskov. Path vector face routing: Geographic routing with local face information. In *Proceedings of ICNP 2005*, pages 147–158, November 2005.
- [21] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of SenSys 2003*, pages 63–75, November 2003.
- [22] J. Newsome and D. Song. GEM: Graph EMbedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of SenSys 2003*, pages 76–88, November 2003.
- [23] C. Perkins. Ad-hoc on-demand distance vector routing. In Proceedings of IEEE MILCOM '97, November 1997.
- [24] C. Perkins and P. Bhagwat. Highly dynamic destinationsequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of ACM SIGCOMM'94 Conference*, pages 234–244, August 1994.
- [25] S. Radhakrishnan, G. Racherla, C. N. Sekharan, N. S. V. Rao, and S. G. Batsell. DST – a routing protocol for ad hoc networks using distributed spanning trees. In *IEEE Wireless Communications* and Networking Conference, 1999.
- [26] A. Rao, C. H. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proceedings of Mobicom 2003*, pages 96–108, San Diego, CA, September 2003.
- [27] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with ght, a geographic hash table. Mobile Networks and Applications (MONET), Journal of Special Issues on Mobility of Systems, Users, Data, and Computing, 2003.
- [28] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage in sensornets. In Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), September 2002.
- [29] K. Seada, A. Helmy, and R. Govindan. On the effect of localization errors on geographic face routing in sensor networks. In *Proceedings of IPSN'04*, pages 71–80, April 2004.
- [30] T. J. Shepard. A channel access scheme for large dense packet radio networks. In *Proceedings of the ACM SIGCOMM '96 Con*ference. ACM SIGCOMM, August 1996.
- [31] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Transactions* on Communications, 32(3):246–257, 1984.
- [32] C. Technlogies. Mica2 series wireless measurement system. http://www.xbow.com.
- [33] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12:261–268, 1980.
- [34] G. Xing, C. Lu, R. Pless, and Q. Huang. On greedy geographic routing algorithms in sensing-covered networks. In *Proceedings* of MobiHoc '04, pages 31–42, May 2004.
- [35] Y. Zhao, B. Li, Q. Zhang, Y. Chen, and W. Zhu. Hop ID based routing in mobile ad hoc networks. In *Proceedings of ICNP 2005*, pages 179–190, November 2005.

Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks

Ben Pfaff, Tal Garfinkel, Mendel Rosenblum {blp,talg,mendel}@cs.stanford.edu

Stanford University Department of Computer Science

Abstract

Virtual disks are the main form of storage in today's virtual machine environments. They offer many attractive features, including whole system versioning, isolation, and mobility, that are absent from current file systems. Unfortunately, the low-level interface of virtual disks is very coarse-grained, forcing all-or-nothing whole system rollback, and opaque, offering no practical means of sharing. These problems impose serious limitations on virtual disks' usability, security, and ease of management.

To overcome these limitations, we offer Ventana, a *virtualization aware file system*. Ventana combines the file-based storage and sharing benefits of a conventional distributed file system with the versioning, mobility, and access control features that make virtual disks so compelling.

1 Introduction

Virtual disks, the main form of storage in today's virtual machine environments, have many attractive properties, including a simple, powerful model for versioning, rollback, mobility, and isolation. Virtual disks also allow VMs to be created easily and stored economically, freeing users to configure large numbers of VMs. This enables a new usage model in which VMs are specialized for particular tasks.

Unfortunately, virtual disks have serious shortcomings. Their low-level isolation prevents shared access to storage, which hinders delegation of VM management, so users must administer their own growing collections of machines. Rollback and versioning takes place at the granularity of a whole virtual disk, which encourages mismanagement and reduces security. Finally, virtual disks' lack of structure obstructs searching or retrieving data in their version histories [34].

Conversely, existing distributed file systems support fine-grained controlled sharing, but not the versioning, isolation, and encapsulation features that make virtual disks so useful.

To bridge the gap between these two worlds, we present

Ventana, a *virtualization aware file system* (VAFS). Ventana extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This attains the benefits of virtual disks, without compromising usability, security, or ease of management.

Unlike traditional virtual disks whose allocation and composition is relatively static, in Ventana storage is ephemeral and highly composable, being allocated on demand as a *view* of the file system. This allows virtual machines to be rapidly created, specialized, and discarded, minimizing the storage and management overhead of setting up a new machine.

We describe the principles behind virtualization aware file systems. We also present our prototype implementation of Ventana and explore the practical benefits that a VAFS offers to VM users and administrators.

We will begin by examining the properties of virtual disks. Section 2 explores their limitations, to motivate our desire for file system-based virtual machine storage, then Section 3 details virtual disks' compelling features. Section 4 shows how to integrate these features into a distributed file system by presenting Ventana, a virtualization aware file system. Section 5 focuses on our prototype implementation of Ventana and Section 6 demonstrates a usage scenario. Sections 7 and 8 discuss related and future work and Section 9 concludes.

2 Motivation

Virtual machines are changing the way that users perceive a "machine." Traditionally, machines were static entities. Users had one or a few, and each machine was treated as general-purpose. The design of virtual machines, and even their name, has largely been driven by this perception.

However, virtual machine usage is changing as users discover that a VM can be as temporary as a file. VMs can be created and destroyed at will, checkpointed and versioned, passed among users, and specialized for particular tasks. Virtual disks, that is, files used to simulate

disks, aid these more dynamic uses by offering fully encapsulated storage, isolation, mobility, and other benefits that will be discussed fully in Section 3.

Before that, to motivate our work, we will highlight the significant shortcomings of virtual disks. Most importantly, virtual disks offer no simple way to share read and write access between multiple parties, which frustrates delegating VM management. At the same time, the dynamic usage model for VMs causes them to proliferate, which introduces new security and management risks and makes such delegation sorely needed [9, 31].

Second, although it is easy to create multiple hierarchical versions of virtual disks, other important activities are difficult. A normal file system is easy to search with command-line or graphical tools, but searching through multiple versions of a virtual disk is a cumbersome, manual process. Deleting sensitive data from old versions of a virtual disk is similarly difficult.

Finally, a virtual disk has no externally visible structure, which forces entire disks to roll back at a time, despite the possible negative consequences [9]. Whether they realize it or not, whole-disk rollback is hardly ever what people actually want. For example, system security precludes rolling back password files, firewall rules, encryption keys, and binaries patched for security, and functionality may be impaired by rolling back network configuration files. Furthermore, the best choice of version retention policy varies from file to file [23], but virtual disks can only distinguish version policies on a whole-disk level.

These limitations of virtual disks led us to question why they are the standard form of storage in virtual environments. We concluded that their most compelling feature is compatibility. All of their other features can be realized in a network file system. By adopting a widely used network file system protocol, we can even achieve reasonable compatibility.

The following section details the virtual disk features that we wish to integrate into a network file system. The design issues raised in this integration are then covered in Section 4.

3 Virtual Disk Features

Virtual disks are, above all, backward compatible, because they provide the same block-level interface as physical disks. This section examines other important features that virtual disks offer, such as versioning, isolation, and encapsulation, and the usage models that they enable. This discussion shapes the design for Ventana presented in the next section.

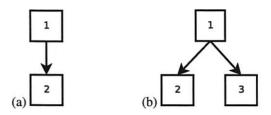


FIGURE 1: Snapshots of a VM: (a) first two snapshots; (b) after resuming again from snapshot 1, then taking a third snapshot.

3.1 Versioning

Because any saved version of a virtual machine can be resumed any number of times, VM histories take the form of a tree. Consider a user who "checkpoints" or "snapshots" a VM, permanently saving the current version as version 1. He uses the VM for a while longer, then checkpoints it again as version 2. So far, the version history is linear, as shown in Figure 1(a). Later, he again resumes from version 1, uses it for a while, then snapshots it another time as version 3. The tree of VMs now looks like Figure 1(b). The user can resume any version any number of times and create new snapshots based on these existing versions, expanding the tree.

Virtual disks efficiently support this tree-shaped version model. A virtual disk starts with an initial or "base" version that contains all blocks (all-zero blocks may be omitted), corresponding to snapshot 1. The base version may have any number of "child" versions, and so may those versions recursively. Thus, like virtual machines, the versions of virtual disks form a tree. Each child version contains only a pointer to its parent and those blocks that differ from its parent. This copy-on-write sharing allows each child version to be stored in space proportional to the differences between it and its parent. Some implementations also support content-based sharing that shares identical blocks regardless of parent/child relationships.

Virtual disk versioning is useful for short-term recovery from mistakes, such as inadvertently deleting or corrupting files, or for long-term capture of milestones in configuration or development of a system. Linear history also effectively supports these usage models. But hierarchical versions offer additional benefits, described below.

Specialization Virtual disks enable versions to be used for specialization, analogous to the use of inheritance in object-oriented languages. Starting from a base disk, one may fork multiple branches and install a different set of applications in each one for a specialized task, then branch these for different projects, and so on. This is easily supported by virtual disks, but today's file systems have no close analogue.

Non-Persistence Virtual disks support "non-persistent storage." That is, they allow users to make temporary changes to disks during a given run of a virtual machine, then throw away those changes once the run is complete. This usage pattern is handy in many situations, such as software testing, education, electronic "kiosk" applications, and honeypots. Traditional file systems have no concept of non-persistence.

3.2 Isolation

Everything in a virtual machine, including virtual disks, exists in a protection domain decoupled from external constraints and enforcement mechanisms. This supports important changes in what users can do.

Orthogonal Privilege With the contents of the virtual machine safely decoupled from the outside world, access controls are put into the hands of the VM owner (often a single user). There is thus no need to couple them to a broader notion of principals. Users of a VM are provided with their own "orthogonal privilege domain." This allows the user to use whatever operating systems or applications he wants, at his discretion, because he is not constrained by the normal access control model restricting who can install what applications.

Name Space Isolation VMs can serve in the same role filled by chroot, BSD jails, application sandboxes, and similar mechanisms. An operating system inside a VM can even be easier to set up than more specialized, OS-specific jails that require special configuration. It is also easier to reason about the security of such a VM than about specialized OS mechanisms. A key reason for this is that VMs afford a simple mechanism for name space isolation, i.e. for preventing an application confined to a VM modifying outside system resources. The VM has no way to name anything outside the VM system without additional privilege, e.g. access to a shared network. A secure VMM can isolate its VMs perfectly.

3.3 Encapsulation

A virtual disk fully encapsulates storage state. Entire virtual disks, and accompanying virtual machine state, can easily be copied across a network or onto portable media, notebook computers, etc.

Capturing Dependencies The versioning model of virtual disks is coarse-grained, at the level of an entire disk. This has the benefit of capturing all possible dependencies with no extra effort from the user. Thus, short-term "undo" using a virtual disk can reliably back out operations with complex dependencies, such as installation or

removal of a major application or device driver, or a complex, automated configuration change.

Full capture of dependencies also helps in saving milestones in the configuration of a system. The snapshot will not be broken by subsequent changes in other parts of the system, such as the kernel or libraries, because those dependencies are part of the snapshot [13].

Finally, integrating dependencies simplifies and speeds branching. To start work on a new version of a project or try out a new configuration, all the required pieces come along automatically. There is no need to again set up libraries or configure a machine.

Mobility A virtual disk can be copied from one medium to another without retaining any tie to its original location. Thus, it can be used while disconnected from the network. Virtual disks thereby offer mobility, the ability to pick up a machine and go.

Merging and handling of conflicts has long been an important problem for file systems that support disconnected operation [16], but there is no automatic means to merge virtual disks. Nevertheless, virtual disks are useful for mobility, indicating that merging is not important in the common case. (In practice, when merging is important, users tend to use revision control systems.)

4 Design

This section describes Ventana, an architecture for a virtualization aware file system. Ventana resembles a conventional distributed file system in that it provides centralized storage for a collection of file trees, allowing transparency and collaborative sharing among users. Ventana's distinction is its versioning, isolation, and encapsulation features to support virtualization, based on virtual disk support for these same features,

The high-level architecture of Ventana can apply to various low-level architectures: centralized or decentralized, block-structured or object-structured, etc. We restrict this section to essential, high-level design elements. The following section discusses specific choices made in our prototype.

We adopt the convention that an operating system inside a virtual machine is a *guest OS*. Ventana's clients run in virtual machines.

Ventana offers the following abstractions:

Branches Ventana supports VM-style versioning with *branches*. A *private branch* is created for use primarily by a single VM, making the branch effectively private, like a virtual disk. A *shared branch* is intended for use by multiple VMs. In a shared branch, changes made from one VM are visible to the others, so these branches can

be used for sharing files, like a conventional network file system.

Non-persistent branches, whose contents do not survive across reboots are also provided, as are volatile branches, whose contents are never stored on a central server, and are deleted upon migration. These features are especially useful for providing storage for caches and cryptographic material that for efficiency or security reasons, respectively, should not be stored or migrated.

Branches are detailed in Section 4.1.

Views Ventana is organized as a collection of file trees. To instantiate a VM, a *view* is constructed by mapping one or more of these trees into a new file system name space. For example, a base operating system, add-on applications, and user home directories might each be mounted from a separate file tree.

This provides a basic model for supporting name space isolation and allows for rapid synthesis of new virtual machines, without the space or management overhead normally associated with setting up a new virtual disk.

Section 4.2 describes views in more detail.

Access Control File permissions in Ventana must satisfy two kinds of needs: those of the guest OSes to partition functionality according to the guests' own principals, and those of users to control access to confidential information. Ventana provides orthogonal types of *file ACLs* to satisfy these needs.

Ventana also offers branch ACLs which support common VM usage patterns, such as one user granting others permission to clone a branch and modify the copy (but not the original), and version ACLs which alleviate security problems introduced by file versioning.

Section 4.3 describes access control in Ventana.

Disconnected Operation Ventana allows for a very simple model of mobility by supporting disconnected operation, through a combination of aggressive caching and versioning. Section 4.4 talks about disconnected operation in Ventana.

4.1 Branches

Some conventional file systems support versioning of files and directories. Details about which versions are retained, when older versions are deleted, and how older versions are named vary. However, in all of them, versioning is "linear," that is, at any point in each file has a unique latest version.

When versions form a tree that grows in more than one direction, asking for the latest version of a file can be ambiguous. The file system must provide a way for users to express where in the tree to look for a file version.

To appreciate these potential ambiguities, consider an example. Ziggy allows Yves, Xena, and Walt to each fork a personalized version of her VM. The version tree for a file personalized by each person would look something like Figure 2(a). If an access to a file by default refers to the latest version anywhere in the tree, then each person's changes would appear in the others' VMs. Thus, the tree of versions would act like a chain of linear versions.

In a different situation, suppose Vince and Uma use a shared area in the file system for collaboration. Most of the time, they do want to see the latest version of a file. Thus, the version history of such a file should be linear, with each update following up on the previous one, resembling Figure 2(b).

The essential difference between these two cases is intention. The version tree alone cannot distinguish between desires for shared or personalized versions of the file system without knowledge of intention.

Consider another file in Ziggy's VM. If only Yves has created a personalized version of the file, then the version tree looks like Figure 2(c). The shape of this tree cannot be distinguished from an early version of Figure 2(b). Thus, Ventana must provide a way for users to specify their intentions.

4.1.1 Private and Shared Branches

Ventana introduces *branches* to resolve the above difficulty. A branch is a linear chain in the tree of versions. Because a branch is linear, it is meaningful to refer to the latest version of a file in a branch, or the version at a particular point in time.

A branch begins as an exact copy of the contents of some other branch at the current time, or at a chosen earlier time. After creation, the new branch and the branch that was copied are independent, so that modifying one has no effect on the other.

Branches are created by copying. Thus, multiple branches may contain the same version of a file. Therefore, for a file access to be unambiguous, both a branch and a file must be specified. Mounting a tree in a virtualization aware file system requires specifying the branch to mount.

If a single client wants a private copy of the file tree, a *private branch* is created for its exclusive use. Like a file system on a virtual disk, a private branch will only be modified by a single client in a single VM, but in other respects it resembles a conventional network file system. In particular, access to files by entities other than the guest that "owns" the branch is easily possible, enabling centralized management such as scanning for malware, file backup, and tracking VM version histories.

If multiple clients mount the same branch of a Ventana file tree, then those clients see a shared view of the files

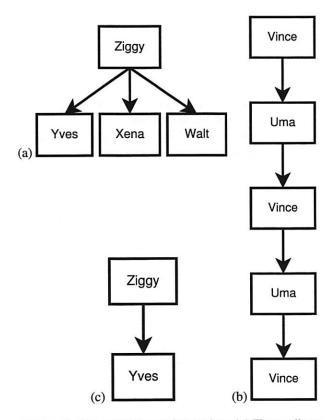


FIGURE 2: Trees of file versions when (a) Ziggy allows Yves, Xena, and Walt to fork personalized versions of his VM; (b) Vince and Uma collaboratively edit a file; and (c) Ziggy's VM has been forked by Yves, as in (a), but not yet by Xena or Walt.

it contains. As in a conventional network file system, a change made by one client in such a *shared branch* will be immediately visible to the others. Of course, propagation of changes between clients is still subject to the ordinary issues of cache consistency in a network file system.

The distinction between shared and private branches is simply the number of clients expected to write to the branch. If necessary, centralized management tools can modify files in a so-called "private" branch (e.g. to quarantine malware) but this is intended to be uncommon. Either type of branch might have any number of read-only clients.

A single file might have versions in shared and private branches. For example, a shared branch used for collaboration between several users might be forked off into a private branch by another user for some experimental changes. Later, the private branch could be discarded or consolidated into the shared branch.

4.1.2 Other Types of Branches

In addition to shared and private branches, there are several other useful qualifiers to attach to file trees.

Files in a *non-persistent branch* are deleted when a VM is rebooted. These are useful for directories of temporary files such as /tmp.

Files in a *volatile branch* are also deleted on reboot. They are never stored permanently on the central server, and are deleted when a VM is migrated from one physical machine to another. They are useful for caches (e.g. /var/cache on GNU/Linux) that need not be migrated and for storing security tokens (e.g. Kerberos tickets) that should not reside on a central server.

Maintaining any version history for some files is an inherent security risk [9]. For example, the OpenSSL cryptography library stores a "random seed" file in the file system. If this is stored in a snapshot, every time a given snapshot is resumed, the same random seed will be used. In the worst case, we will see the same sequence of random numbers on every execution. Even in the best case, its behavior may be easier to predict, and if old versions are kept, then it may be possible to guess past behavior (e.g. keys generated in past runs).

Ventana offers *unversioned files* as a solution. Unversioned files are never versioned, whether linearly or in a tree. Changes always evolve monotonically forward with time. Applications for unversioned files include storing cryptographic material, firewall rules, password files, or any other configuration state where rollback would be problematic.

4.2 Views

Ventana is organized as a set of file trees, each of which contains related files. For example, some file trees might contain root file systems for booting various operating systems (Linux, Windows XP, ...) and their variants (Debian, Red Hat, SP1, SP2, ...). Another might contain file systems for running various local or specialized applications. A third would have a hierarchy for each user's files.

Creating a new VM mainly requires synthesizing a *view* of the file system for the VM. This is accomplished by mapping one or more trees (or parts of trees) into a new namespace. For example, the Debian root file system might be combined with a set of applications and user home directories. Thus, OSes, applications, and users can easily "mix and match" in a Ventana environment.

Whether each file tree in a view is mounted in a shared or a private branch depends on the user's intentions. The root file system and applications could be mounted in private branches to allow the user to update and modify his own system configuration. Alternately, they could be mounted in shared branches (probably read-only) to allow maintenance to be done by a third party. In the latter case, some parts of the file system would still need to be private, e.g. /var under GNU/Linux. Home directories would likely be shared, to allow the user to see a con-

sistent view of his and others' files regardless of the VM viewing them.

4.3 Access Control

Access control is different in virtual disks and network file systems. The guest OS controls every byte on a virtual disk. It is responsible for tracking ownership and permissions and making access control decisions in the file system. The virtual disk itself has no access control responsibility. A VAFS cannot use this scheme, because allowing every guest OS to access any file, even those that belong to other VMs, is obviously unacceptable. There must be enough control in the system to prevent abuse.

Access control in a conventional network file system is the reverse of the situation for a virtual disk. The file server is ultimately in charge of access control. As a network file system client, a guest OS can deny access to its own processes, but it cannot override the server's refusal to grant access. Commonly, NFS servers deny access as the superuser ("squash root") and CIFS and AFS servers grant access only via principals authenticated to the network.

This style of access control is also, by itself, inappropriate in a VAFS. Ventana should not deny a guest OS control over its own binaries, libraries, and applications. If these were, for example, stored on an NFS server configured to "squash root," the guest OS would not be able to create or access any files as the superuser. If they were stored on a CIFS or AFS server, the guest OS would only be able to store files as users authenticated to the network. In practice this would prevent the guest from dividing up ownership of files based on their function (system binaries, print server, web server, mail server, ...), as many systems do.

Ventana solves the problem of access control through multiple types of ACLs: *file ACLs*, *version ACLs*, and *branch ACLs*. For any access to be allowed, it must be permitted by all three applicable ACLs. Each kind of ACL serves a different primary purpose. The three types are described individually below.

4.3.1 File ACLs

File ACLs provide protection on files and directories that users conventionally expect and OSes conventionally provide. Ventana supports two types of file ACLs that provide orthogonal privileges. *Guest file ACLs* are primarily for guest OS use. Guest OSes have the same level of control over guest file ACLs that they do over permissions in a virtual disk. In contrast, *server file ACLs* provide protection that guest OSes cannot bypass, similar to permissions enforced by a conventional network file server.

Both types of file ACLs apply to individual files. They are versioned in the same way as other file metadata. Thus, revising a file ACL creates a new version of the file with the new file ACL. The old version of the file continues to have the old file ACL.

Guest file ACLs are managed and enforced by the guest OS using its own rules and principals. Ventana merely provides storage. These ACLs are expressed in the guest OS's preferred form. We have so far implemented only the 9-bit rwxrwxrwx access control lists used by the Unixlike guest OSes. Guest file ACLs allow the guest OS to divide up file privileges based on roles.

Server file ACLs, the other type of file ACL, are managed and enforced by Ventana and stored in Ventana's own format. Server file ACLs allow users to control access to files across all file system clients.

4.3.2 Version ACLs

A version ACL applies to a version of a file. They are stored as part of a version, not as file metadata, so that changing a version ACL does not create a new file version. Every version of a file has an independent version ACL. Conversely, when multiple branches contain the same version of a file, that single version ACL applies in each case. Version ACLs are not versioned themselves. Like server file ACLs, version ACLs are enforced by Ventana itself.

Version ACLs are Ventana's solution to a class of security problem common to all versioning file systems. Suppose Terry creates a file and writes confidential data to it. Soon afterward, Terry realizes that the file's permissions incorrectly allow Sally to read it, so he corrects the permissions. In a file system without versioning, the file would then be safe from Sally, as long as she had not already read it. If the permissions on older file versions are fixed, however, Sally can still access the older version of the file.

A partial solution to Terry's problem is to grant access to older versions based on the current version's permissions, as Network Appliance filers do [32]. Now, suppose Terry edits a file to remove confidential information, then grants read permission to Sally. Under this rule, Sally can then view the older, confidential versions of the file, so this rule is also flawed.

Another idea is to add a permission bit to each file's metadata that determines whether a user may read a file once it has been superseded by a newer version, as in the S4 self-securing storage system [27]. Unfortunately, modifying permissions creates a new version (as does any change to file metadata) and only the new version is changed. Thus, this permission bit is effective only if the user sets it before writing confidential data, so it would not protect Terry.

Only two version rights exist. The "r" (read) version

right is Ventana's solution to Terry's problem. At any time, Terry can revoke the read right on old versions of files he has created, preventing access to those file versions. The "c" (change) right is required to change a version ACL. It is implicitly held by the creator of a version. (Any given file version is immutable, so there is no "write" right.)

4.3.3 Branch ACLs

A branch ACL applies to all of the files in a particular branch and controls access to current and older versions of files. Like version ACLs, branch ACLs are accessed with special tools and enforced by Ventana.

The "n" (newest) branch right permits read access to the latest version of files in a branch. It also controls forking the latest version of the branch.

In addition to "n", the "w" (write) right is required to modify any files within a branch. A user who has "n" but not "w" may fork the branch. Then, as owner of the new branch, he may change its ACL and modify the files in the new branch. This does not introduce a security hole because the user may only modify the files in the new branch, not those in the old branch. The user's access to files in the new branch are, of course, still subject to Ventana file ACLs and version ACLs.

The "o" (old) right is required to access old versions of files within a branch. This right offers an alternate solution to Terry's problem of insecure access to old versions. If Terry controls the branch in which the old versions were created, then he can use its branch ACL to prevent other users from accessing old versions of any file in the branch. This is thus a simpler but less focused approach than adjusting the appropriate version ACL.

The "c" (change) right is required to change a branch ACL. It is implicitly held by the owner of a branch.

4.4 Disconnected Operation

Virtual disks can be used while disconnected from the network, as long as the entire disk has been copied onto the disconnected machine. Thus, for a virtualization aware file system to be as widely useful as a virtual disk, it must also gracefully tolerate network disconnection.

Research in network file systems has identified a number of features required for successful disconnected operation [16, 15, 12]. Many of these features apply to Ventana in the same way as conventional network file systems. Ventana, for example, can cache file system data and metadata on disk, which allows it to store enough data and metadata to last the period of disconnection. Our prototype caches entire files, not individual blocks, to avoid the need to allow reading only part of a file during disconnection, which is surprising at best. Ventana can also

buffer changes to files and directories and write them back upon reconnection. Some details of these features of Ventana are included in the description of our prototype (see Section 5).

Handling conflicts, that is, different changes to the same files, is a thorny issue in a design for disconnected operation. Fortunately, earlier studies of disconnection have shown conflicts to be rare in practice [16]. In Ventana conflicts may be even rarer, because they cannot occur in private branches. Therefore, Ventana does not try to intelligently handle conflicts. Instead, changes by disconnected clients are committed at the time of reconnection, regardless of whether those files have been changed in the meantime by other clients. If manual merging is needed in shared branches, it is still possible based on old versions of the files. To make it easy to identify file versions just before reconnection, Ventana creates a new branch just before it commits the disconnected changes.

5 Prototype

To show that our ideas can be realized in a practical and efficient way, we developed a simple prototype of Ventana. This section describes the prototype's design and use.

The Ventana prototype is written in C. We developed it under Debian GNU/Linux "unstable" on x86 PCs running Linux 2.6.x, using VMware Workstation 5.0 as VMM. The servers in the prototype run as Linux user processes and communicate over TCP using the GNU C library implementation of ONC RPC [26].

Figure 3 outlines Ventana's structure, which is described in more detail below.

5.1 Server Architecture

A conventional file system operates on what Unix calls a block device, that is, an array of numbered blocks. Our prototype is instead layered on top of an *object store* [10, 7]. An object store contains *objects*, sparse arrays of bytes numbered from zero to infinity, similar to files. In the Ventana prototype, objects are immutable.

The object store consists of one or more *object servers*, each of which stores some of the file system's objects and provides a network interface for storing new objects and retrieving the contents of old ones. Objects are identified by randomly selected 128-bit integers called *object numbers*. Object numbers are generated randomly to allow them to be chosen without coordination between hosts. Collisions are unlikely as long as significantly fewer than 2⁶⁴ have been generated, according to the "birthday paradox" [25].

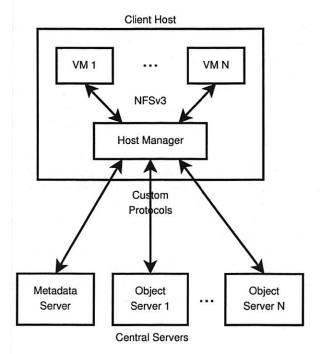


FIGURE 3: Structure of Ventana. Each machine whose VMs use Ventana runs a host manager. The host manager talks to the VMs over NFSv3 and to Ventana's centralized metadata and object servers over a custom protocol.

Each version of a file's data or metadata is stored as an object. When a file's data or metadata is changed, the new version is stored as a new object under a new object number. The old object is not changed and it may still be accessed under its original object number. However, this does not mean that every intermediate change takes up space in the object store, because client hosts (that is, machines that run Ventana clients in VMs) consolidate changes before they commit a new object.

As in an ordinary file system, each file is identified by an inode number, which is again a 128-bit, randomly selected integer. Each file may have many versions across many branches. When a client host needs to know what object stores the latest version of a file in a particular branch, it consults the *version database* by contacting the *metadata server*. The metadata server maintains the version database that tracks the versions of each file, the *branch database* that tracks the file system's branch structure, the database that associates branch names and numbers, and the database that stores VM configurations.

5.2 Client Architecture

The *host manager* is the client-side part of the Ventana prototype. One copy of the host manager runs on each platform and services any number of local client VMs.

Our prototype does not encapsulate the host manager itself in a VM.

For compatibility with existing clients, the host manager includes a NFSv3 [2] server for clients to use for file access. NFSv3 is both easy to implement and widely supported, even on Windows (with Microsoft's free Services for Unix).

The host manager maintains in-memory and on-disk caches of file system data and metadata. Objects may be cached indefinitely because they are immutable. Objects are cached in their entirety to simplify implementing the prototype and to enable disconnected operation (see Section 5.2.3). Records in the version and branch databases are also immutable, except for the ACLs they include, which change rarely. In a shared branch, records added to the version database to announce a new file version are a cache consistency issue, so the host manager checks the version database for new versions on each access (except when disconnected). In a private branch, normally only one client modifies the branch at a time, so that client's host manager can cache data in the branch for a long time (or until the client VM is migrated to another host), although other hosts should check for updates more often.

The host manager also buffers file writes. When a client writes a file, the host manager writes the modified file to the local disk. Further changes to the file are also written to the same file. If the client requests that writes be committed to stable storage, e.g. to allow the guest to flush its buffer cache or to honor an fsync call, then the host manager commits the modified files to the local disk. Commitment does not perform a round trip on a physical network.

5.2.1 Branch Snapshots

After some amount of time, the host manager takes a snapshot of outstanding changes within a branch. Users can also explicitly create (and optionally name) branch snapshots. A snapshot of a branch is created simply by forking of the branch, which has the desired effect because forking a branch copies its content. In fact, copying occurs on a copy-on-write basis, so that the first write to any of the files in the snapshot creates and modifies a new copy of the file. Creating a branch also inserts a record in the branch database.

After it takes a snapshot, the host manager uploads the objects it contains into the object store. Then, it sends records for the new file versions to a metadata server, which commits them to the version database in a single atomic transaction. The changes are now visible to other clients.

The host manager assumes that private branch data is relatively uninteresting to clients on other hosts, so it takes snapshots in private branches relatively rarely (every 5 minutes). On the other hand, other users may be actively using files in shared branches, so the host manager takes snapshots often (every 3 seconds).

Because branch snapshots are actually branches themselves, older versions of files can be viewed using regular file commands by first adding the snapshot branch to the view in use. Branches created as snapshots are by default read-only, to reduce the chance of later confusion if a file's "older version" actually turns out to have been modified.

5.2.2 Views and VMs

Multiple branches can be composed into a view. Ventana describes a view with a simple text format that resembles a Unix fstab, e.g.:

```
debian:/ / shared,ro
home-dirs:/ /home shared
bob-version:/ /proj private
```

Each line describes a mapping between a branch, or a subset of a branch, and a directory within the view. We say that each branch is *attached* to its directory in the view.

A VM comprises a view, plus configuration parameters for networking, system boot, and so on. A VM could be described by the view above followed by these additional options:

```
-pxe-kernel debian:/boot/vmlinuz
-ram 64
```

Ventana provides a utility to start a VM based on such a specification. Given the above VM specification, it would set up a network boot environment (using the PXE protocol) to boot the kernel in /boot/vmlinuz in the debian branch, then launch VMware Workstation for the user to allow the user to interact with the VM.

VM Snapshots Ventana supports snapshots of VMs just as it does snapshots of branches.² A snapshot of a VM is a snapshot of each branch in the VM's view combined with a snapshot of the VM's runtime state (RAM, device state, ...). To create a snapshot, Ventana snapshots the branches included in the VM, copies the runtime state file written by Workstation into Ventana as an unnamed file, and saves a description of the view and a pointer to the suspend file.

Later, another Ventana utility may be used to resume from the snapshot. When a VM snapshot is resumed, private branches have the contents that they did when the snapshot was taken, and shared branches are up-to-date. Ventana also allows resuming with a "frozen" copy of shared branches as of the time of the snapshot. Snapshots can be resumed any number of times, so resuming forks each private branch in the VM for repeatability.

5.2.3 Disconnected Operation

The host manager supports disconnected operation, that is, file access is allowed even without connectivity to the metadata and object server. Of course, access is degraded during disconnection: only cached files may be read, and changes in shared branches by clients on the other hosts are not visible. Write access is unimpeded. Disconnected operation is implemented in the host manager, not in clients, so all clients support disconnected operation.

We designed the prototype with disconnected operation in mind. Caching eliminates the need to consult the metadata and object servers for most operations, and ondisk caching allows for a large enough cache to be useful for extended disconnection. Whole-object caching avoids surprising semantics that would allow only part of a file to be read. Write buffering allows writing back changes to be delayed until reconnection.

We have not implemented user-configurable "hoarding" policies in the prototype. Implementing them as described by Kistler et al. [16] would be a logical extension.

6 Usage Scenario

This section presents a scenario for use of Ventana and shows how, in this setting, Ventana offers a better solution than both virtual disks and network file systems.

6.1 Scenario

We set our scene at Widgamatic, a manufacturer and distributor of widgets.

6.1.1 Alice the Administrator

Alice is Widgamatic's system administrator in charge of virtual machines. Software used at Widgamatic has diverse requirements, and Widgamatic's employees have widely varying preferences. Alice wants to accommodate everyone as much as she can, so she supports various operating systems: Debian, Ubuntu, Red Hat, and SUSE distributions of GNU/Linux, plus Windows XP and Windows Server 2003. For each of these, Alice creates a shared branch and installs the base OS and some commonly used applications. She sets the branch ACLs to allow any user to read, but not write, these branches.

Alice creates common, a second shared branch, to hold files that should be uniform company-wide, such as

¹We use "attach" instead of "mount" because mounts are implemented inside an OS, whereas the guest OS that uses Ventana does not implement and is not aware of the view's composition.

²VMware Workstation has its own snapshot capability. Ventana's snapshot mechanism demonstrates VM snapshots might be integrated into a VAFS.

```
ubuntu:/
                                                                     shared, ro
                       home-dirs:/
                                                   /home
                                                                     shared
                            none
                                                   /tmp
                                                                     non-persistent
12ff2fd27656c7c7e07c5ea1e2da367f:/var
                                                   /var
                                                                     private
                        cad-soft:/
                                                   /opt/cad-soft
                                                                     shared, ro
                          common:/etc/resolv.conf /etc/resolv.conf shared,ro
                          common:/etc/passwd
                                                   /etc/passwd
                                                                     shared, ro
8368e293a23163f6d2b2c27aad2b6640:/etc/hostname
                                                   /etc/hostname
                                                                     private
b6236341bd1014777cla54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

FIGURE 4: Partial specification of the view for Bob's basic VM.

```
carl-debian:/ / private
home-dirs:/ /home shared
none /tmp non-persistent
common:/etc/resolv.conf /etc/resolv.conf shared,ro
common:/etc/passwd /etc/passwd shared,ro
b6236341bd1014777c1a54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

FIGURE 5: Partial specification of the view for Carl's custom VM.

/etc/hosts and /etc/resolv.conf. Again, she sets branch ACLs to grant other users read-only access.

Alice also creates a shared branch for user home directories, called home-dirs, and adds a directory for each Widgamatic user in the root of this branch. Alice sets the branch ACL to allow any user to read or write the branch, and server file ACLs so that, by default, each user can read or write only his (or her) home directory. Users can of course modify server file ACLs in their home directories as needed.

6.1.2 Bob's Basic VM

Bob is a Widgamatic user with basic needs. Bob uses a utility written by Alice to create a Linux-based VM primarily from shared branches. Figure 4 shows part of the specification written by this utility.

The root of Bob's VM is attached to the Ubuntu shared branch created by Alice. This branch's ACL prevents Bob modifying files in the branch (it is attached read-only besides). The Linux file system is well suited for this situation, because its top-level hierarchies segregate files based on whether they can be attached read-only during normal system operation. The /usr tree is an example of a hierarchy that normally need not be modifiable.

The /home and /tmp trees are the most prominent examples of hierarchies that must be writable, so Bob's VM attaches a writable shared branch and a non-persistent branch, respectively, at these points. Keyword none in place of a branch name in /tmp's entry causes an initially empty branch to be attached.

The filename/var hierarchy must be writable and persistent, and it cannot be shared between machines. Thus, Al-

ice's utility handles /var by creating a fork of the Ubuntu branch, then attaching the forked branch's /var privately in the VM. The utility does not give the forked branch a name, so the VM specification gives the 128-bit branch identifier as 32 hexadecimal digits.

Bob needs to use the company's CAD software to design widgets, so the CAD software distribution is attached into his VM.

Most of the VM's configuration files in /etc receive their contents from the Ubuntu branch attached at the VM's root. Some, such as /etc/resolv.conf and /etc/passwd shown here, are attached from Alice's "common files" branch. This allows Alice to update a file in just that branch and have the changes automatically reflected in every VM. A few, such as /etc/hostname shown here, are attached from private branches to allow their contents to be customized for the particular VM. Finally, data that should not be versioned at all, such as the private host key used to identify an SSH server, is attached from an unversioned branch. The latter two branches are, like the /var branch, unnamed.

Bob's VM, and VMs created in similar ways, would automatically receive the benefits of changes and updates made by Alice as soon as she made them. They would also see changes made by other users to their home directories as soon as they occur.

6.1.3 Carl's Custom VM

Carl wants more control over his VM. He prefers Debian, which is available as a branch maintained by Alice, so he can base his VM upon Alice's. Carl forks a private branch from Alice's Debian branch and names the new branch

carl-debian.

Carl integrates his branch into a VM of his own, using a specification that in part looks like Figure 5. Carl could write this specification by hand, or he might choose to start from one, like Bob's, generated by Alice's utility. Using a private branch as root directory means that Carl need not attach private branches on top of /var or /etc/hostname, making Carl's specification shorter than Bob's.

Even though Carl's base operating system is private, Carl's VM still attaches many of the same shared branches that Bob's VM does. Shared home directories and common configuration files ease Carl's administrative burden just as they do Bob's. He could choose to keep private copies of these files, but to little obvious benefit.

Carl bears more of the burden of his own system administration, because Alice's changes to shared branches do not automatically propagate to his private branch. Carl could use Ventana to observe how the parent debian branch has changed since the fork, or Alice could monitor forked branches to ensure that important patches are applied in a timely fashion.

6.1.4 Alice in Action

One morning Alice reads a bulletin announcing a critical security vulnerability in Mozilla Firefox. Alice must do her best to make sure that the vulnerable version is properly patched in every VM. In a VM environment based on virtual disks, this would be a daunting task. Ventana, however, reduces the magnitude of the problem considerably.

First, Alice patches the branches that she maintains. This immediately fixes VMs that use her shared branches, such as Bob's VM.

Second, Alice can take steps to fix others' VMs as well. Ventana puts a spectrum of options at her disposal. Alice could do nothing and assume that Bob and Carl will act responsibly. She could scan VMs for the insecure binary and email their owners (she can even check up on them later). She could patch the insecure binaries herself. Finally, she has many options for denying access to copies of the insecure binary: use a server file ACL to deny reading or executing it, use a Ventana version ACL to prevent reading it even as the older version of a file, use a branch ACL to deny any access to the branch that contains it (perhaps appropriate for long-unused branches), and so on. Alice can take these steps for any file stored in Ventana, whether contained in a VM that is powered on or off or suspended, or even if it is not in any VM or view at all.

Third, once the immediate problem is solved, Alice can work to prevent its future recurrence. She can configure a malware scanner to examine each new version of a file added to Ventana as to whether it is the vulnerable program and, if so, alert Alice or its owner (or take some other action). Thus, Alice has reasonable assurance that if this particular problem recurs, it can be quickly detected and fixed.

6.2 Benefits for Widgamatic

We now consider how Alice, Bob, Carl, and everyone else at Widgamatic benefit from using Ventana instead of virtual disks. We use virtual disks as our main basis of comparison because Ventana's advantages over conventional distributed file systems are more straightforward: they are the versioning, isolation, and encapsulation features that we intentionally added to it and have already described in detail.

6.2.1 Central Storage

It's easy for Bob or Carl to create virtual machines. When virtual disks are used, it's also easy for Bob or Carl to copy them to a physical machine or a removable medium, then lose or forget about the machine or the medium. If the virtual machine is rediscovered later, it may be missing fixes for important security problems that have arisen in the meantime.

Ventana's central storage makes it more difficult to lose or entirely forget about VMs, heading off the problem before it occurs. Other dedicated VM storage systems also yield this benefit [30, 31].

6.2.2 Looking Inside Storage

Alice's administration tasks can benefit from "looking inside" storage. Consider backup. Bob and Carl want the ability to restore old versions of files, but Alice can easily back up virtual disks only as a collection of disk blocks. Disk blocks are opaque, making it hard for Bob or Carl even to determine which version of a virtual disk contains the file to restore. Doing partial backups of virtual disks, e.g. to exclude blocks from deleted temporary files or paging files, is also difficult.

File-based backup, partial backup, and related features can be implemented for virtual disks, but only by mounting the virtual disk or writing code to do the equivalent. In any case, software must have an intimate knowledge of file system structures and must be maintained as those structures change among operating systems and over time. Mounting an untrusted disk can itself be a security hole [24].

On the other hand, Ventana's organization into files and directories gives it a higher level of structure that makes it easy to look inside a Ventana file system. Thus, file-based backup and restore requires no special effort in Ventana.

(Of course, in Ventana it is natural to use versioning to access file "backups" and ensure access by backing up Ventana servers' storage.)

6.2.3 Sharing

Sharing is an important feature of storage systems. Bob and Carl might wish to collaborate on a project, or Carl might ask Alice to install some software in his VM for him. Virtual disks make sharing difficult. Consider how Alice could access Carl's files if they were stored on a virtual disk. If Carl's VM were powered on or suspended, modifying his file system would risk the guest OS's integrity, because the interaction with the guest's data and metadata caches would be unpredictable. Even reading Carl's file system would be unreliable while it was changing, e.g. consider the race condition if a block from a deleted directory was reused to store an ordinary file block.

On the other hand, Ventana gives Alice full read and write access to virtual machines, even those that are online or suspended. Alice can examine or modify Carl's files, whether the VM or VMs that use them are running, suspended, or powered off, and Bob and Carl can work together on their project without introducing any special new risks.

6.2.4 Security

If Widgamatic's VMs were stored in virtual disks, Alice would have a hard time scanning them for malware. She could request that users run a malware scanner inside each of their VMs, but it would be difficult for her to enforce this rule or ensure that the scanner was kept up-to-date. Even if Bob and Carl carefully followed her instructions, VMs powered on after being off for a long time would be susceptible to vulnerabilities discovered in the meantime until they were updated.

Ventana allows Alice to deploy a scanner that can examine each new version of a file in selected branches, or in all branches. Conversely, when new vulnerabilities are found, it can scan old versions of files as well as current versions (as time is available). If malware is detected in Bob's branch, the scanner could alert Bob (or Alice), delete the file, change the file's permission, or remove the virus from the file. (Even in a private branch, files may be externally modified, although it takes longer for changes to propagate in each direction.)

Ventana provides another important benefit for scanners: the scanner operates in a protection domain separate from any guest operating system. When virtual disks store VMs, scanners normally run as part of the guest operating system because, as we've seen, even read-only access to active virtual disks has pitfalls. But this allows a "root

kit" to subvert the guest operating system and the malware scanner in a single step. If Alice runs her scanner in a different VM, it must be compromised separately. Alice could even configure the scanner to run in non-persistent mode, so rebooting it would temporarily relieve any compromise, although of course not the underlying vulnerability.

A host-based intrusion detection system could use a "lie detector" test that compares the file system seen by programs running inside the VM against the file system in Ventana to detect root kits, as in LiveWire [8].

6.2.5 Access to Multiple Versions

Suppose Bob wants to look at the history of a document he's been working on for some time. He wants to retrieve and compare all its earlier versions. One option for Bob is to read the old versions directly from older versions of the virtual disk, but this requires accurate interpretation of the file system, which is difficult to maintain over time. A more likely alternative for Bob is to resume or power on each older version of the VM, then use the guest OS to copy the file in that old VM somewhere convenient. Unfortunately, this can take a lot of time, especially if the VM has to boot, and every older version takes extra effort.

With Ventana, Bob can attach all the older versions of his branch directly to his view. After that, the different versions can be accessed with normal file commands: diff to view differences between versions, grep to search the history, and so on. Bob can also recover older versions simply by copying them into the his working branch.

7 Future Work

Ventana demonstrates the principles behind a VAFS, but many important issues remain to be explored, such as scalability and performance. We have measured Ventana's performance to be competitive with other user-level NFS servers in most cases with simple branching. However, deep chains of branches seem to introduce the need for compromise between storage efficiency and file lookup performance.

Storage reuse is another area for further work. The Ventana prototype does not have any mechanism for deleting data. We have not yet found a way to efficiently support both creation of branches and the determination that an object is no longer in use in any branch.

8 Related Work

Parallax [31] demonstrates that virtual disks can be stored centrally with very high scalability. Parallax allows vir-

tual disks to be efficiently used and modified in a copyon-write fashion by many users. Unlike Ventana, it does not allow cooperative sharing among users, nor does it enhance the transparency or improve the granularity of virtual disks.

VMware ESX Server includes the VMFS file system, which is designed for storing large files such as virtual disks [30]. VMFS allows for snapshots and copy-on-write sharing, but not the other features of a virtualization aware file system.

Live migration of virtual machines [4] requires the VM's storage to be available on the network. Ventana, as a distributed file system particularly suited to VM storage, provides a reasonable approach.

Whitaker et al. [33, 34] used whole-system versioning to mechanically discover the origin of a problem by doing binary search through the history of a system. They note the "semantic gap" in trying to relate changes to a virtual disk with higher-level actions. We believe that a VAFS, in which changes to files and directories may be observed directly, could help to reduce this semantic gap.

The Ventana prototype of course has much in common with other file systems. Object stores are an increasingly common way to structure file systems [10, 7, 28]. Objects in Ventana are immutable, which is unusual among object stores, although in this respect Ventana resembles the Cedar file system and, more recently, EMC's Centera system [11, 6]. PVFS2, a network file system for high-bandwidth parallel file I/O, is another file system that uses Berkeley DB databases to store file system metadata [21].

Many versioning file systems exist, in research systems such as Cedar, Elephant, and S4, and in production systems such as WAFL (used by Network Appliance filers) and VMS [11, 23, 27, 14, 18]. A versioning file system on top of a virtual disk allows old versions to be easily accessed inside the VM, but does not address the other downsides of virtual disks. None of these systems supports the tree-structured versions necessary to properly handle the natural evolution of virtual machines. The version retention policies introduced in Elephant might be usefully applied to Ventana.

Online file archives, such as Venti, also support accessing old versions of files, but again only linear versioning is supported [22].

Ventana's tree-structured version model is related to the model used in revision control systems, such as CVS [3]. A version created by merging versions from different branches has more than one parent, so versions in revision control systems are actually structured as directed acyclic graphs. Revision control systems would generally not be good "back end" storage for Ventana or another VAFS because they typically store only a single "latest" version of a file for efficient retrieval. Retrieving other versions, including the latest version of files in branches other than

the "main branch," requires application of patches [29]. Files marked "binary," however, often include each revision in full, without using patches, so use of "binary" files might be an acceptable choice.

Vesta [13] is a software configuration management system whose primary file access interface is over NFS, like Ventana. Dependency tracking in Vesta allows for precise, high-performance, repeatable builds. Similar tracking by a VAFS might enable better understanding of which files and versions should be retained over the long term.

We proposed extending a distributed file system, which already supports fine-grained sharing, by adding versioning that supports virtual machines. An alternative is to allow virtual disks, which already support VM-style versioning, to support sharing by adding a locking layer, as can be done for physical disks [19, 1]. This approach requires committing to a particular on-disk format, which makes changes and extensions more difficult. It also either requires each client to understand the disk format, which is a compatibility issue, or use of a network proxy that does understand the format. In the latter case the proxy is equivalent to Ventana's host manager, and storage underlying it is really an implementation detail.

A "union" or "overlay" file system [20, 17] can stack a writable file system above layers of read-only file systems. If the top layer is the current branch and lower layers are the branches that it was forked from, something like tree versioning can be obtained. The effect is imperfect because changes to lower layers can "show through" to the top. Symbolic link farms can also stack layers of directories, often for multi-architecture software builds [5], but link farms are not transparent to the user or software.

9 Conclusion

Ventana is a *virtualization aware* distributed file system that provides the powerful versioning, security, and mobility properties of virtual disks, while overcoming their coarse-grained versioning and their opacity that frustrates cooperative sharing. This allows Ventana to support the rich usage models facilitated by virtual machines, while avoiding the security pitfalls, management difficulties, and usability problems that virtual disks suffer from.

We believe that virtualization aware file systems have an important role to play in the evolution of virtual machines from their physical machine inspired roots, toward being a more lightweight, flexible, and general-purpose mechanism for organizing systems.

Acknowledgements

This work was supported in part by the National Science Foundation under award 0121481 and by TRUST (Team for Research in Ubiquitous Secure Technology), which also receives support from the National Science Foundation under award CCF-0424422. We would like to thank Carl Waldspurger, Tim Mann, Jim Chow, Paul Twohey, Junfeng Yang, Joe Little, our shepherd Steve Hand, and the anonymous reviewers for their comments.

References

- R. C. Burns. Data Management in a Distributed File System for Storage Area Networks. PhD thesis, University of California Santa Cruz, March 2000.
- [2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995.
- [3] P. Cederqvist et al. Version Management with CVS, 2005.
- [4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In 2nd Symposium on Network Systems Design and Implementation. USENIX, 2005.
- [5] P. Eggert. Multi-architecture builds using GNU make. http://make.paulandlesley.org/multi-arch.html, August 2000.
- [6] EMC Corporation. EMC Centera content addressed storage system. http://www.emc.com/products/systems/centera.jsp, October 2005.
- [7] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In 2nd International IEEE Symposium on Mass Storage Systems and Technologies, Sardinia, Italy, July 2005.
- [8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proc. Network and Distributed Systems Security Symposium, February 2003.
- T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In 10th Workshop on Hot Topics in Operating Systems, May 2005
- [10] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS), pages 272–284, New York, NY, USA, 1997. ACM Press.
- [11] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.
- [12] J. S. Heidemann, T. W. Page, Jr., R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In Workshop on the Management of Replicated Data, pages 2–5, 1992.
- [13] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Research Report 168, Compaq Systems Research Center, March 2001.
- [14] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Technical report, Network Appliance, 1995.
- [15] L. Huston and P. Honeyman. Disconnected operation for AFS. In First Usenix Symposium on Mobile and Location-Independent Computing, pages 1–10, August 1994.
- [16] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. ACM Transactions on Computer Systems, 10(1):3–25, February 1992.
- [17] M. Klotzbuecher. mini_fo: The mini fanout overlay file system. http://www.denx.de/twiki/bin/view/Know/MiniFOHome, October 2005.

- [18] K. McCoy. VMS file system internals. Digital Press, Newton, MA, USA, 1990.
- [19] T. McGregor and J. Cleary. A block-based network file system. In 21st Australasian Computer Science Conference, volume 20 of Australian Computer Science Communications, pages 133–144, Perth, February 1998. Springer.
- [20] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In Summer UKUUG Conference, pages 1–9, London, July 1990.
- [21] PVFS2: Parallel virtual file system 2. http://www.pvfs.org/pvfs2, October 2005.
- [22] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In FAST '02: Proceedings of the Conference on File and Storage Technologies, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.
- [23] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In 17th ACM Symposium on Operating Systems Principles, pages 110–123, New York, NY, USA, 1999. ACM Press.
- [24] C. Sar, P. Twohey, J. Yang, C. Cadar, and D. Engler. Discovering malicious disks with symbolic execution. In *IEEE Symposium on Security and Privacy*, May 2006.
- [25] B. Schneier. Applied Cryptography. Wiley, 2nd edition, 1996.
- [26] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Aug. 1995.
- [27] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In 4th USENIX Symposium on Operating System Design and Implementation, pages 165–180, 2000.
- [28] C. F. Systems. Lustre. http://lustre.org/.
- [29] W. F. Tichy. RCS—a system for version control. Software Practice and Experience, 15(7):637–654, 1985.
- [30] VMware ESX Server. http://www.vmware.com/products/esx, October 2005.
- [31] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In 10th Hot Topics in Operating Systems. USENIX, May 2005.
- [32] A. Watson, P. Benn, A. G. Yoder, and H. T. Sun. Multiprotocol data access: NFS, CIFS, and HTTP. Technical report, Network Appliance, 2005.
- [33] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In 6th Symposium on Operating Systems Design and Implementation, December 2004.
- [34] A. Whitaker, R. S. Cox, and S. D. Gribble. Using time travel to diagnose computer problems. In 11th ACM SIGOPS European Workshop, Leuven, Belgium, September 2004.

Olive: distributed point-in-time branching storage for real systems

Marcos K. Aguilera Susan Spence Alistair Veitch Hewlett-Packard Laboratories, Palo Alto, California, USA

Abstract. This paper describes Olive, the first distributed block storage system to provide consistent pointin-time branching. Point-in-time branching allows users to recursively and quickly snapshot or clone the storage state. It has a wide range of applications including testing new deployments or upgrades without disrupting a running system, quickly provisioning large homogeneous systems, and preserving old versions of data. Olive provides block-level access and strong consistency for broad applicability, allowing it to branch file systems, database systems, and every other storage application that ultimately stores data on block storage. Olive is distributed and replicated to provide fault tolerance and availability. Providing strong consistency for branching in a replicated distributed system is a technical challenge that we address in this work. We evaluate Olive and show that branching typically takes a few tens of milliseconds, and so it has little impact on I/O's.

1 Introduction

Distributed replicated storage systems provide many benefits over single-server solutions, like better scalability and cheaper reliability. Scalability comes from dividing work among many servers, and cheaper reliability comes from replicating over unreliable commodity hardware instead of using specialized fault-tolerant hardware. Distribution, however, poses challenges: variable asymmetric network delays, and the inability to distinguish a node that has crashed from one that is slow to respond, make it impossible for nodes to always be consistent with each other. These inconsistencies must be dealt with, ideally in a way transparent to users.

This paper proposes a new scheme to provide *point-in-time branching* of storage, or the ability to recursively fork off storage branches that can evolve independently, in a replicated distributed system. Branching storage includes two basic functions: snapshots and clones. A *snap-shot* is a read-only virtual copy that preserves the state of storage at a given point in time, while a *clone* is a virtual copy that can change independently of its source. ¹ These storage branches are recursive, meaning that branches can be created off other branches.

Branching storage has many uses that become more important as the size of storage increases without a corresponding increase in data transfer rates-a trend that has made it increasingly difficult to manipulate ever larger data volumes. As an example application, suppose that a user wishes to install and test a software upgrade without disturbing the current working version. Without branching storage, this could involve copying large amounts of application data and environment state, which can take hours or days. With branching storage, the user can simply create a storage clone very quickly, and install the upgrade on the clone, without disturbing the original version. As another application, suppose that an administrator wishes to provision storage to many homogeneous computers from a "golden" copy, as is often needed in a computer lab, store, or data center. Without branching storage, this involves copying entire storage volumes many times. With branching distributed storage, the administrator can simply clone the golden copy once for each computer. Besides being fast to create, clones are space efficient because they share common unmodified blocks.

This paper describes Olive, a novel point-in-time branching storage system that is efficient, distributed, broadly applicable, and fault tolerant. Providing point-in-time branching for distributed replicated systems raises new consistency issues because of the need to simultaneously coordinate replicas and capture distributed state when there are many outstanding operations. We believe our techniques to handle these issues are applicable not just to Olive, but also to other distributed replicated storage systems.

While designing Olive, our goal was to maximize its applicability to real systems. To do so, we made two broad design choices: (1) Provide branching at the lowest level: block storage. Branching functionality can be designed at various levels, including database systems, file systems, object stores, or block storage. By choosing block storage, Olive can be used to branch file systems, database systems, or any application that ultimately stores data on block storage. (2) Preclude changes to storage clients. Changes to storage clients are a huge inhibitor for adoption of new storage solutions, due to the large existing base of clients and applications. Thus, Olive does not

¹Clones are sometimes called "writable snapshots", but we avoid this term since it is an oxymoron.

require storage clients to be modified in any way: they do not need to run special protocols, install special drivers, or use special storage paradigms. In fact, we took this goal to an extreme, by showing that Olive can support an industry-standard storage protocol, iSCSI [22]. Olive presents clones and snapshots as regular block volumes on a network.

Olive provides replicated storage distributed over a network for fault tolerance. We use quorum-based data replication for high-availability, which provides three benefits. First, the system requires only a quorum (e.g., a majority) of replicas to be accessible to a client at any time. Second, the accessible quorum can vary with different clients, due to their placement in the network for example, and over time, due to transient network behavior or brick failures. Third, if a client cannot access a quorum, part of the storage becomes temporarily unavailable to that client, but neither storage integrity nor other clients are affected.

To broaden applicability, Olive also provides a very strong form of consistency, linearizability [9], which allows Olive to be used by applications that are very demanding on storage consistency, in addition to less demanding applications. Roughly speaking, linearizability requires that operations appear to occur at a single point in time between the start and end of the operation. For branching storage, this means that if a clone or snapshot is requested at time t_1 and completes at time $t_2 > t_1$ then it will capture the global state of storage at a single point in time between t_1 and t_2 . Note that linearizability implies other forms of consistency used in storage systems, like sequential consistency, causal consistency, and crash consistency. Crash consistency means that a branch initially captures some state that could have resulted from a crash, which is important because applications typically know how to recover from such states.

We implemented Olive within the Federated Array of Bricks (FAB), a low-cost distributed storage system that provides block-level storage and uses distributed data replication for fault tolerance [3, 20]. We show that branching a volume typically takes a few tens of milliseconds so it has little impact on I/O's. Experiments also validate the consistency of our scheme.

In summary, Olive is the first distributed block storage system to provide point-in-time branching. This is achieved without any client changes, which is an important consideration for applicability to real systems. A key contribution of Olive is its scheme to provide strong consistency for distributed replicated storage, by ensuring that replication and branching are coordinated carefully. As far as we know, Olive is the first system to tackle this issue.

This paper is organized as follows. In Section 2 we explain the assumed environment and the requirements for Olive. Section 3 explains the version tree, a simple struc-

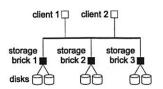


Figure 1: Distributed storage setting.

ture that is used both internally and by users of Olive. We explain the exact consistency that Olive provides in Section 4. Olive's efficiency comes from sharing data between branches; we explain how this is done in Section 5. Section 6 covers the main algorithms in Olive, which are responsible for providing consistency. Section 7 has a discussion on data layout on physical storage. We describe the evaluation of Olive in Section 8 and related work in Section 9.

2 Environment and requirements

Environment. We consider a distributed system where nodes communicate with each other by sending point-to-point messages over network links. Every pair of nodes can send messages to each other. Nodes may fail by crashing; a crashed node stops executing and becomes unresponsive. We do not consider malicious failures. Network links may fail by dropping messages, but we assume that if a message is repeatedly sent and the destination does not crash then the destination eventually receives the message (no permanent partitions). We do not assume that the network is synchronous or that network delays are bounded.

Some nodes in the network are storage nodes or *bricks*; together, they implement the storage system. Other nodes have clients running storage applications, like file systems or database systems (Figure 1). In an ideal world, storage clients can execute custom protocols to read and write data, and these protocols can implement replication for fault tolerance. But in practice, deploying custom protocols at clients is very difficult; clients instead use standard storage protocols to send a read or write request to a *single* brick. This brick can use custom protocols to execute the client request, and then returns the result to the client, if any, using the standard storage protocol.

Requirements. Our goal in producing Olive is to obtain a distributed block-based storage system that provides point-in-time branching. Distributed means that Olive is implemented by multiple storage nodes, and it is usable by multiple application nodes. Block-based storage means that storage is accessed through fixed-length units called blocks, typically with 512 bytes each. Olive is expected to have the usual attributes of a general-purpose storage service: good reliability, availability, and performance. Providing point-in-time branching means to implement two functions: *snapshots* and *clones* of a storage volume. A snapshot of a volume is a data collection that *retains* the past contents of a volume despite updates.

Snapshots are useful for archiving data or in other situations where old versions of data are needed. For broad usability, we require that a snapshot be accessible as a storage volume, so that applications that run on regular storage can also run on snapshots. The volume from which a snapshot originates is called its *source*.

A clone of a volume is another volume that starts out as a virtual copy but may later diverge, allowing data to evolve in multiple concurrent ways. For flexibility, we require all volumes to be clonable, including clones and snapshot volumes themselves. Cloning a volume results in a new writable volume whose initial contents are identical to the source volume. While the data in the clone can change, the snapshot retains its original data. It is possible to clone a snapshot multiple times, for example to experiment with different evolutions of data from the same snapshot.

We also make the following further requirements of Olive:

- Do not require client changes. For broadest applicability, clients should use a standard network storage protocol for reading and writing. There are currently no standard interfaces for requesting snapshots and clones, so we allow some flexibility here, but the interface should be minimal and intuitive, so that it can be replaced with a standard interface once it emerges.
- Provide a strong form of consistency. A storage volume may be cloned or snapshotted while there are outstanding I/O operations, because storage clients, such as file systems, can have concurrent activities, and most cannot be expected to pause their activities when a snapshot is made (and requiring this would violate the previous requirement). For broadest applicability, in these cases snapshots and clones should provide a strong form of consistency so that the service can be used with all applications. Roughly speaking, a consistent snapshot ensures that the data that it preserves reflects the state of storage at some point in the past. And a consistent clone ensures that its initial state is a consistent snapshot. We later describe the exact consistency guarantees provided by Olive.
- Avoid performance disruptions. When creating new branches or when using data volumes that are branches, the performance of the storage should not suffer significant impact.
- Be space efficient. Storage branches may share lots of common data, and in those cases, the system should avoid having multiple physical copies in storage.

3 The version tree

The version tree is a simple data structure that Olive uses to describe the relationship between various storage

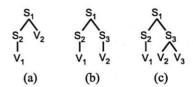


Figure 2: Examples of version trees.

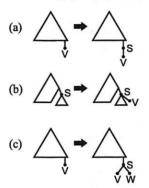


Figure 3: How branching operations affect the version tree: (a) Taking a snapshot S of a writable volume V. (b) Making a clone V of a snapshot S. (c) Making a clone W of a writable volume V.

branches, where each branch is a volume. In the context of Olive, a volume is a set of blocks, but in general it could be any set of data objects that are branched together. Nodes in the version tree correspond to volumes, where a leaf node corresponds to a writable volume, and the ancestors of the leaf node correspond to all its snapshots, with the most recent snapshots closer to the leaf. Inner nodes in the tree are always (read-only) snapshot volumes. Figure 2 (a) is an example of such a tree. There are two leaves, V_1 and V_2 , which are writable volumes, and two snapshots of V_1 , S_1 and S_2 . S_1 is also a snapshot of V_2 . This case might occur if V_2 is created as a clone of S_1 . Figure 2 (b) shows how the tree changes if a user takes a snapshot S_3 of V_2 : S_3 is the parent of V_2 because S_3 is V_2 's most recent snapshot. And Figure 2 (c) shows what happens if the user subsequently creates a clone of

In general, taking a snapshot of a writable volume V results in replacing V with a new node S and adding V as a child of S. This is depicted in Figure 3 (a), where the triangles represent the version trees before and after taking a snapshot. Cloning a snapshot S results in creating a new child V of S (Figure 3 (b)). Cloning a writable volume V corresponds to creating a snapshot of V and then cloning the snapshot. The result is that V is replaced with a snapshot S and two children, V and a new node W (Figure 3 (c)). Note that snapshot S is a by-product of cloning V; the reason for this will become clear later, but roughly speaking it is because we want both V and W to initially share allocation of their data, allowing for space efficiency.

The version tree allows a user to visualize all previous

states of a given writable volume V, as well as to understand where in the past two volumes have diverged. As we will see later, the version tree is also needed by Olive to maintain data consistency.

4 Consistency provided

Intuitively, a replicated storage system that supports branching needs to maintain two forms of consistency:

- Replica consistency: ensuring that replicas have compatible states; and
- Branching consistency: ensuring that snapshots and clones have meaningful contents.

Replica consistency is a form of consistency that is internal to the storage system, while branching consistency is visible to storage clients.

The branching consistency provided by Olive is linearizability [9], a strong and well-understood condition used often in concurrent systems. Roughly speaking, linearizability considers operations that have non-zero durations, and requires each operation to appear to take place at a single point in time, and this point must be between the start and the end of the operation. For branching storage, by definition the operations are read, write, clone, and take snapshot; the start of an operation is when a client requests the operation, and the end is when the client receives a response or acknowledgement. For example, Figure 4 (a) shows a timeline where time flows to the right. There are four operations: three writes to volume V for blocks B_1 , B_2 , and B_3 with data x, y, and z, respectively, and one clone operation on V. These operations have start and end times represented by the endpoints of the lines below each operation. Linearizability requires operations to appear to take effect at a single point on these lines. Figures 4 (b), (c), and (d) show some points in time where each operation could appear to occur in accordance with linearizability. In (b), the clone operation "happens" at a point after the write to B_1 but before the other writes; as a result, the clone incorporates the first write but not the others. In (c) and (d), the clone operation happens at a different place and so the clone incorporates different sets of writes. All these behaviors are allowed by linearizability. One behavior not allowed by linearizability is for the clone to incorporate the writes to B_1 and B_3 , but not the one to B_2 , because there is no way to assign points to each operation to produce this behavior.

Linearizability captures the intuition that if two operations are concurrent then they may be ordered arbitrarily, while if they are sequential then they must follow real time ordering.

Olive achieves branching consistency and replica consistency by building upon an existing protocol for replicated storage [3, 20]. This protocol solves the problem of how new writes are propagated to the replicas, and how reads reconcile data from possibly divergent

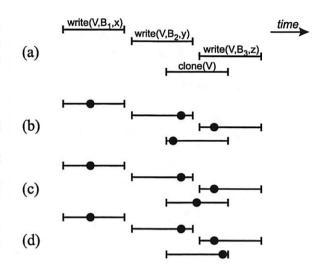


Figure 4: Depiction of linearizability. (a) shows an execution history with three writes and a clone operation. (b)-(d) show three allowable ways to linearize this execution.

replicas, without any branching. The problems that we solve are how to propagate information about new storage branches, how this information interacts with reads, writes and the replicated storage protocol to achieve linearizable branching, and how and when different branches can share data for efficiency. This is explained later in Section 6.

Relation to other forms of consistency. Linearizability is similar to sequential consistency [13], but different because sequential consistency allows an operation to appear to take place after the operation has completed or before the operation has begun. For example, with sequential consistency, the clone could exclude all writes in Figure 4 (a), i.e., the clone appears to occur at a point before all writes. This could occur with an implementation that did not see frequent writes because they are still in some buffer; this implementation, however, would not satisfy linearizability.

Linearizability implies *crash consistency*. Roughly speaking, crash consistency is a consistency condition for snapshots (or clones) that requires that the state captured by a snapshot be one that could have resulted from halting the system unexpectedly (crash). For this definition to be precise, one needs to define what are the allowable states produced by a crash, but typically it means that completed writes are incorporated while outstanding writes may be incorporated partly. It is not difficult to show that linearizability implies this property, in the sense that a branching storage system satisfying linearizability ensures crash consistency of its snapshots or clones.

Crash consistency means that a branch initially captures some state that could have resulted from a crash, which is important because applications typically know how to recover from such states. The recovery procedure typically involves writing data to the volume, and might

require user choices, such as whether to delete unattached inodes, and so it is performed at a clone derived from a snapshot.

5 Data sharing

Storage volumes that are related by branching may have lots of common data, which can share the storage medium. This provides not just space efficiency, but also time efficiency since sharing allows branches to be created quickly by simply manipulating data structures. We next explain these data structures. Throughout this paper, we use the standard terminology that *logical* offsets or blocks are relative to the high-level storage volume, whereas *physical* offsets or blocks are relative to the storage medium (disks).

Each storage node or brick needs logical-to-physical maps that indicate where each logical address of a volume is mapped. This is a map from (volume, logical-offset) to (disk, physical-offset). Because it takes too much space to maintain this map on a byte-per-byte basis, the map is kept at a coarser granularity in terms of disk allocation units, which are chunks of L consecutive bytes where L is some multiple of 512 bytes. L is called the disk allocation size, and it provides a trade-off between flexibility of allocation and the size of the map. It is also useful to think in terms of the reverse physical-to-logical map, which indicates the volume and logical offset that correspond to each disk and physical offset. This map is one-to-many, because storage volumes may share physical blocks. The sharing list of a physical block B is the result of applying this map to block B: it indicates the set of all storage volumes that share B (strictly speaking, the map also indicates the logical offset where the sharing occurs, but this offset is the same for all volumes sharing B). When a write occurs to a block that is being shared, the sharing is broken and the sharing list shrinks. Sharing can be broken in two ways: either the volume being written gets allocated a new block B' (move-on-write), or the volume being written retains B while the old contents of B are copied to B' (copy-on-write). In the common case, there will be exactly one volume V in the sharing list of B or B' (the volume where the write occurs) and the other list will be equal to the original list minus V (the volumes that should preserve the contents before the write). However, there are situations in which the split will result in more than one volume in both B and B'. Those situations are due to recovery reads, which we will cover in the next section.

6 Algorithm

A key contribution of Olive is its algorithm for reading and writing data while providing strong consistency and supporting branching. To explain the algorithm, we first provide some background on quorum-based data replication in Section 6.1; this is not a novelty of Olive. The

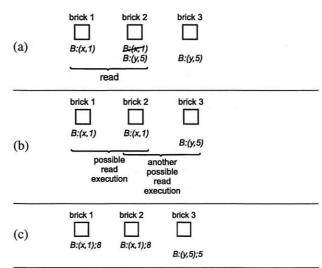


Figure 5: (a) Using timestamps to resolve replica divergence. (b) Non-determinism that arises from partial writes. (c) Largest seen timestamp, shown after semicolons.

novelty is how to provide branching storage on top of the replication scheme, as we explain in Sections 6.2–6.8. Due to lack of space, this paper does not have proofs of correctness; they will be provided in the full version.

6.1 Quorum-based data replication

To tolerate failures, Olive uses quorum-based replication [5, 2, 15, 16, 6] modified to work with real distributed storage systems [3, 20], which we now explain. Storage is replicated at many storage nodes or bricks, such that data is accessible if a quorum of bricks are operational and accessible through the network. In this paper, a quorum means a majority, but other types of quorums are possible [4]. Majorities can vary with time because of variance in network delays and brick load, causing one or another brick to be temporarily slow to respond, or because of brick crashes. To write new data, a coordinator propagates the data with a timestamp to a majority of bricks; the timestamp comes from the coordinator's local clock, which is nearly synchronized with other coordinators' clock most of the time. To read data, a coordinator queries the data at a majority of bricks and decides which data is the most recent using the timestamp. Because any two majorities intersect, at least one brick returns to the coordinator the most recently written data. Figure 5(a) shows an example. Three bricks store data for block B; other blocks are not shown. Initially, data x with timestamp 1 is stored at bricks 1 and 2, a majority; later, data y with timestamp 5 is stored at bricks 2 and 3, another majority; later, a read gets data from bricks 1 and 2, and y is chosen since it has a higher timestamp.

Partial writes. A partial write occurs when the coordinator crashes while writing to some block B, causing the new data to be propagated to only a minority of replicas.

The system is left in a non-deterministic state, because a subsequent read may return either new or old value, depending on whether the majority that reads intersect the minority that wrote. For example, Figure 5(b) shows y at a minority of bricks. A subsequent read will return different values depending on which majority responds first, as determined by network and other delays: if the majority is bricks 1 and 2, the read returns x, but if the majority is bricks 2 and 3, the read returns y due to its higher timestamp. This could lead to the problem of oscillating reads: as majorities change over time, consecutive reads return different values even though there are no writes. To prevent this, the coordinator executes a repair phase, in which it writes back or propagates the value read to a majority of replicas with a new timestamp. In the example, the repair phase writes back x or y with a higher timestamp, say 8, to a majority of bricks.²

If some coordinator writes while another coordinator reads, the repair phase of the read may obliterate an ongoing write. In Figure 5(b), y may be at a minority of bricks not because the coordinator crashed, but because it has not yet finished propagating y; as both write and read coordinators continue to execute, the write back of x may obliterate the write of y. This problem is addressed through an initial announce phase, in which a coordinator announces to a majority of bricks the timestamp that it wants to use; each brick remembers the largest announced timestamp. Thus, writes execute two phases: announce and propagate (Figure 6). For reads, the announce phase can be combined with querying the data at bricks, so reads also execute two phases: announce+query and propagate. Each phase may involve a different majority of bricks. In the second phase, if a coordinator propagates a value with a smaller timestamp than the largest announced timestamp at a brick, the brick rejects the value and returns an error, which causes the coordinator to return an error to the client. Typical clients (e.g., an operating system) then retry the operation, for a few times. Figure 5(c) shows the largest announced timestamp for block B at each brick after the semicolons; this timestamp is different from the timestamp of the data, shown in parenthesis. Bricks 1 and 2 have been announced timestamp 8 of an ongoing read that will later write back x with this timestamp. Meanwhile, an ongoing write with timestamp 5 has propagated y to brick 3; when it tries to propagate y to bricks 1 or 2, an error will occur because these bricks saw timestamp 8. This will cause a client to retry writing y. More generally, the announce phase helps to deal with a stale timestamp: if some value at a brick has timestamp T then T has been announced at a majority of bricks, and so a coordinator

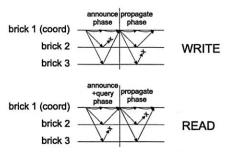


Figure 6: Two-phase write and read operation.

that tries to execute with a smaller timestamp gets an error. The announce phase also allows an important simple optimization for reading: in the first phase, if all bricks return the same data with same timestamp, and indicate that no higher timestamp has been announced, then the repair phase is not needed since the data already has the highest timestamp among all bricks including those that did not respond.

In the above description, the coordinator reads or writes a single block, but the scheme allows operation on a range of blocks, by packaging together information for multiple blocks in the messages of the protocol.

We now explain how branching works with quorumbased storage replication, by using the version tree (Section 3) to determine what content should continue to be shared after a write or a read.

6.2 Using the version tree to determine sharing

Recall that the sharing list of a physical block B is the set of all storage volumes that share B. The sharing list changes over time as new data gets written to volumes. For example, consider the version tree in Figure 2 (c), and suppose that logical block b of volumes S_1 , S_3 , V_2 and V_3 are sharing the same physical block B. Then, the sharing list for B is $\{S_1, S_3, V_2, V_3\}$. If a user writes new data to block b of volume V_3 then a new physical block b is allocated for volume b (assuming move-on-write instead of copy-on-write), and the sharing list for b is reduced to $\{S_1, S_3, V_2\}$; the sharing list created for b is $\{V_3\}$.

Read-only snapshots may get their blocks updated too, because of the repair phase of reads. For example, if the sharing list for B is $\{S_1, S_3, V_2, V_3\}$ and there is a read on snapshot S_3 that requires writing back to S_3 then a new physical block B' is allocated for the data being written back (assuming move-on-write) and the sharing list for B' is set to $\{S_3, V_2, V_3\}$, while the sharing list for B gets reduced to $\{S_1\}$.

The general rule for splitting a sharing list L is that the volume V being written (or written back) and all its children in L should share the newly written contents, while the other volumes in L should share the old contents This is consistent with the fact that descendants of node V represent later versions of that node, and so if there is a

²Some quorum-based replication schemes write back x or y with its original timestamp, which prevents oscillation from y to x, but still allows one oscillation from x to y, thus allowing a failed write to take effect at an unpredictable arbitrary time in the future. This violates the limited effect property [1], and so Olive does not employ such schemes.

change on V to fix nondeterminism then descendants of V that are sharing data with V also need to fix the nondeterminism in the same way.

6.3 Achieving consistency

Recall that Olive must provide two forms of consistency: replica consistency and branching consistency. We now explain the details of the new mechanisms used to achieve them; keep in mind that each form of consistency cannot be provided in isolation, but rather require a single scheme that provides both. Thus, the separation of techniques below is merely for didactic purposes.

Replica consistency. To create a new storage branch, a user sends a request to one of the bricks, say c. Brick c decides how the version tree needs to be updated, based on the type of the new branch as explained earlier, and then propagates this update to other bricks. This propagation is done with uniform reliable broadcast [7, 8], which ensures that if one brick receives the update then all live bricks also receive it, despite failures, thus ensuring eventual consistency. Olive uses a simple algorithm to implement uniform reliable broadcast, described in Section 6.8.

While the propagation is happening, however, bricks will have divergent version trees. For example, if we take a new snapshot of V_2 from the situation in Figure 2 (a), bricks will eventually arrive at the tree in Figure 2 (b), where S_3 is the new snapshot. If a new write occurs, and all bricks have the tree in (b), then the write results in a copy-on-write on all replicas to preserve the contents for snapshot S_3 . But what happens if the new snapshot is still propagating, and some bricks have (a), while others have (b)?

In our scheme, the coordinator for the write decides what to do, and the replicas just follow that decision. We implement this by having a version number associated with writable volumes; this number is incremented every time the volume gets a new snapshot. The number is the depth of the volume's node in the tree if snapshots are not deleted, but could be higher if snapshots are deleted. For example, Figure 7 shows a version tree with two writable volumes V_1 and V_2 (leaves), assuming no snapshots have been deleted. The current version of V_1 and V_2 is 3; version 2 of V_1 is snapshot S_2 , while version 1 is snapshot S_1 . When executing a write on a volume, the coordinator reads the volume's version according to its local view; bricks receiving a write from the coordinator use that number to decide where the write gets stored. For example, if the coordinator decides to write version 2 of volume V_2 (because the coordinator's version tree is slightly out of date and does not have snapshot S_3 yet), then a brick that has snapshot S_3 will store the new data by overwriting data for S_3 rather than doing a copy-on-write. This ensures that replicas treat all writes consistently.

Branching consistency. To achieve branching consistency, the coordinator of a write checks that the version

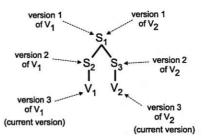


Figure 7: Version numbers associated with *writable* volumes V_1 and V_2 for a given version tree.

number that it wants to pick is the same at a majority of bricks. If not, the coordinator picks the highest version number seen and retries, until successful—this is called the version-retry technique, which is key to obtaining consistency. This process is piggybacked on the first phase of the two-phase write protocol (from Section 6.1), and so it has little additional cost if the coordinator does not have to retry, which is the common case. In the second phase of the two-phase write, which is when the data actually gets written to each brick, the coordinator tells bricks which version k it picked along with the data and timestamp to be written. When a brick receives this information, it stores the data in the appropriate physical block according to the logical-to-physical map. If that block is being shared with many volumes, the sharing may have to be broken according to the general rule in Section 6.2.

The above technique, whereby the coordinator retries the first phase until a majority of replica bricks have identical version numbers, effectively delays writes while a snapshot is taken. This is different from the well-known but simplistic technique of pausing I/O's during a snapshot, in which the coordinator acts in three phases: it first tells bricks to pause their I/O's, then it tells bricks that branching has occurred, and finally it tells bricks to resume I/O's. This simplistic technique is slow because there are three sequential phases, where each phase requires all bricks (not just a majority) to acknowledge before moving to the next phase. Requiring all bricks to respond eliminates the benefits of quorums. In contrast, with our scheme only a quorum of bricks needs to respond, and we embed the necessary delays within the write protocol without the need for explicit pause and resume actions. The result is less time to take snapshots (and hence smaller write delays during snapshots), and less complex handling of failures: with the simplistic scheme, there has to be a way to resume paused bricks if the coordinator fails, whereas with our scheme uniform reliable broadcast ensures that the snapshot information eventually propagates to the live replicas, regardless of failures, and so a write does not get stuck.

For branch consistency, reads also need to be handled carefully. First, recall that reads use a repair phase to fix nondeterministic state that arise from partial writes. For example, consider Figure 5(b), where there is a partial write of y to block B in volume V, and assume that a subsequent read coordinator obtains responses from bricks 1 and 2, and so the coordinator picks x as the value to be read; the coordinator will then write back x to a majority of bricks with a new higher timestamp. How do we perform this write back with branching storage? Each of the three bricks have a sharing list for block B, and for consistency, it is important that the writing back of x occurs not just at volume V, but also at all ancestor volumes of V in the version tree, that appear in the sharing list for block B at the brick that returned x to the read coordinator.

For example, suppose that there are three bricks, brick 1, brick 2 and brick 3, with version tree as in Figure 2 (a), and some logical block B has value x, which is shared between volumes S_1 and V_2 at all bricks. Now suppose there is a write for B in volume V_2 with data y, but the write is partial and only reaches brick 3, due to a failure of the write coordinator. Thus, at brick 3, the sharing of B has been broken, but this is not so at the other bricks. Now suppose that a new snapshot of V_2 is taken resulting in the version tree as in Figure 2 (b) at all bricks.

The resulting situation for logical block B is that brick 3 has data x for S_1 and data y for $\{S_3, V_2\}$, while brick I and brick 2 have data x for $\{S_1, S_3, V_2\}$. Now suppose there is a read for B in volume V_2 . While executing the read, suppose brick 1 and brick 2 respond to the coordinator, but brick 3 is slow. Then x is picked as the value being read, and there is a write back of x for volumes S_1 , S_3 , and V_2 with a new timestamp. This causes brick 3 to restore back the sharing between S_1, S_3 and V_2 . It also causes all bricks to adopt the new timestamp for B in volumes S_1, S_3 and V_2 , not just for V_2 . The reason is to ensure that y cannot be read for any of these volumes; in fact, when the system resolves the nondeterminism for V_2 by deciding that the failed write of y never occurred then it must make a consistent decision for the previous snapshots S_1 and S_3 .

6.4 Creating new snapshots and clones

We now explain how Olive creates snapshots of writable volumes, clones of snapshots, and clones of writable volumes.

Creating a snapshot of a volume V is a very simple operation: it simply requires updating the version tree and incrementing the version number of V at a majority of bricks. This is done using uniform reliable broadcast, to ensure that the updates are propagated regardless of failures. The brick creating a snapshot waits until a majority of bricks have acknowledged the updates before telling the user that the operation is completed. This is necessary because reads to the snapshot should be prohibited until a majority of bricks have received the update: otherwise, two reads to the same snapshot could return different data (this could happen if a write occurs to the volume being

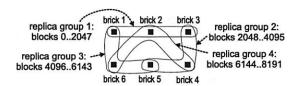


Figure 8: Replica groups storing different blocks.

snapshotted).

To create a clone of a snapshot S, a brick updates the version tree, propagates the update using uniform reliable broadcast, and waits for a majority of acknowledgements.

To create a clone of a writable volume V, a brick simply creates a snapshot S of V and then creates a clone of S, using the above procedures.

Note that if two clients take snapshots simultaneously of the same volume, there is a chance that both get the same snapshot. This is not problematic since snapshots are read-only. As for clones, it is desirable to actually create multiple clones, and so the coordinator adds a unique identifier to clones, namely, an id for the coordinator plus an increasing number.

6.5 Multiple replica groups

So far, we have been assuming that all storage blocks are replicated across all bricks. Instead, if there are many bricks, it may be desirable to replicate blocks at some of the bricks in a way that spreads load, as shown in Figure 8. The set of bricks that replicate a block is called a *replica group*. A real system with many bricks will have many replica groups, and in general they need not intersect.

To snapshot a volume, bricks at all replica groups must coordinate to ensure that branching takes effect atomically. Our algorithm is designed so that, while the relatively infrequent snapshot operation must contact a majority of bricks in every replica group, the more common read and write operations only has to contact the replica group of the block involved. To do this, we introduce the notion of a *stable* version: version v of a volume is stable iff the current version is at least v at a majority of bricks in all replica groups. The algorithm ensures that if a brick reads or writes using version v then v is stable. Note that if v is stable then no operations in any replica group will use a version smaller than v, because of the version-retry technique (Section 6.3). This provides consistency across replica groups.

A variable *stableVersion* keeps the largest version that a brick knows to be stable, and it is updated as follows. When a coordinator takes a snapshot of a volume, it uses a *uniform confirmed broadcast* to ensure that a volume's *stableVersion* is only incremented after a majority of bricks in every replica group of the volume has incremented their *currentVersion*. Roughly speaking, uniform confirmed broadcast ensures that (1) either all correct bricks (in all replica groups) deliver a message, or none

do, and (2) if the broadcaster does not fail then all correct bricks get the message. It also provides a *confirmation* of the broadcast through the primitive *confirm-deliver*. Roughly speaking, confirm-deliver(m) informs a brick that message m has been delivered at a quorum of bricks, where quorum in our system means a majority of bricks in each replica group. Uniform confirmed broadcast ensures that (1) either all bricks get confirm-deliver(m), or none do, and (2) if some brick delivers m, then all correct bricks get confirm-deliver(m). We show how to implement uniform confirmed broadcast in Section 6.8, using a simple 4-phase protocol.

When a brick delivers a message with a new version, it updates its *currentVersion* variable. When a brick gets a confirmation for this message, it updates its *stableVersion* variable. This ensures that a version *currentVersion* is stable when *stableVersion* > *currentVersion*.

When a coordinator starts a read or write on a volume, it initially waits until its version *currentVersion* for the volume is stable. It does not need to contact other bricks to determine this, because it keeps track of versions already known to be stable using the *stableVersion*, as described above. In the common case, the coordinator should not even have to wait, since *currentVersion* is likely to be stable already. Once *currentVersion* is stable, the read or write operation is executed on this version of the volume.

It is worth noting that, while a 4-phase protocol for uniform confirmed broadcast may slow down the snapshot operation, the reads and writes are only delayed during one of those 4 phases: after a message with a new version is received but before its confirmation (this is the time when *currentVersion* is not stable). Given that taking snapshots are not as frequent as performing I/O, the 4-phase protocol does not severely impact the system as a whole. This is confirmed by our experiments.

6.6 Multi-volume branching

Sometimes it is useful to clone or snapshot many volumes simultaneously and atomically—an operation that we call *multi-volume branching*. For example, a database system may store the log and tables in separate volumes, which is a typical scenario. In that case, if the table and log volumes are cloned separately then the cloned log may be out of sync with the cloned tables. With multi-volume branching, the cloning of two or more volumes can occur atomically, thus ensuring consistency between them. In terms of linearizability, this means that the cloning of all volumes appear to take place at a single point in time, rather than having a different point for each volume.

Olive provides multi-volume branching using exactly the same mechanisms as replica groups: stable versions and uniform confirmed broadcast. A snapshot or clone operation uses uniform confirmed broadcast to contact all bricks that serve the volumes being branched. The broadcast carries new version numbers for each volume being branched. When the message is delivered, it causes a brick to increment the *currentVersions* of the volumes, causing a brief delay on new writes to those volumes. Soon after, the confirmation of delivery makes those versions stable, allowing new writes to continue.

6.7 Deleting storage branches

A user deletes a volume by sending a request to one of the bricks, who acts as the coordinator, as for other storage operations. The coordinator reliably broadcasts the request to all bricks.

Upon receipt of the request, a brick p does the following. It first removes the volume from the sharing list of all physical blocks. If the sharing list has become empty for a physical block, the block is marked as free. Brick p then updates the version tree by marking the volume's node as deleted, but the node is not yet removed from the tree, for two reasons: First, the node may have children that are not deleted, and so it should remain in the tree while the children are there. Second, even if the node has no children, another coordinator may be trying to branch the volume while it is being deleted. The actual removal of nodes from the tree happens through a periodic pruning where entire branches are removed: a node is only removed if all its children are marked deleted. This periodic pruning is done with a two-phase protocol that quits after the first phase if any node to be pruned is being branched.

6.8 Implementing uniform confirmed broadcast

Figure 9 gives an algorithm for uniform confirmed broadcast, by using point-to-point messages. (It is also easy to modify the algorithm to implement uniform reliable broadcast instead, by replacing the number 4 with 2 in lines 1 and 12 and removing lines 8 and 9.) It works as follows. To broadcast a message, a brick proceeds in 4 phases. In each phase, the brick sends the message and phase number to all bricks, and waits to receive acknowledgements from a quorum of bricks (where in our system, a quorum means a majority of bricks in each replica group). When a brick receives a message from a phase, it sends back an acknowledgement to the sender. In addition, if the phase is 2, the brick delivers the message, and if the phase is 3, the brick confirms the message.

If a brick receives a message for phases 1, 2, or 3, but does not receive a message for the following phase after a while, then the brick suspects that the sender has failed, and takes over the job of the sender (lines 10–14). In practice, one should add some random delay to the checking for this condition; Otherwise, if all bricks check at the same time, then lots of bricks will take over the job of the sender. This will not affect correctness, but may result in lots of messages and extra delay.

7 Data layout on physical storage

When mapping logical addresses to physical storage, it is desirable to preserve locality by placing adjacent logical

```
To broadcast(m):
    for i \leftarrow 1 to 4 do
        send \langle NEW, i, m \rangle to all
        wait to receive (NEW-REP, i, m) from a majority from each replica group
  upon receive \langle NEW, i, m \rangle from q
     send \langle NEW-REP, i, m \rangle to q
     if i = 2 and has not yet delivered m
     then deliver m
     if i = 3 and has not yet confirmed m
Task check
     repeat periodically forever
        if for some m and j \leq 3, received (NEW, j, m) long ago, but
11
           never received (NEW, j+1, m) then
           for i \leftarrow j to 4 do
12
             send \langle \text{NEW}, i, m \rangle to all
13
              wait to receive (NEW-REP, i, m) from a majority
```

Figure 9: Implementation of uniform confirmed broadcast, used for handling multiple replica groups or for multi-volume branching.

addresses in adjacent physical addresses. It can be impossible to simultaneously preserve locality, share data between volumes, and ensure that writes execute efficiently. Indeed, efficient writing to branching storage involves using move-on-write or copy-on-write, but both schemes destroy locality of either the branched volume or source volume.

This issue, however, has little to do with the distributed or replicated nature of storage: it also applies to centralized systems, like disk arrays. In Olive, we did not come up with new solutions to this problem, but just ensured enough flexibility to support existing solutions. Indeed, Olive allows general maps of logical-to-physical storage (Section 5), and so it can support the following:

- Prioritize writable volumes over snapshot volumes, by using copy-on-write instead of move-on write. This preserves locality of the former to the detriment of the latter.
- Use volume priorities to choose between copy-onwrite or move-on-write to preserve locality of the higher priority volume.
- Allocate branched blocks near the original, to preserve locality of all volumes; for example, leave an empty disk track between tracks of data, if space allows, and then use the empty track for move-on-write or copy-on-write.
- Defragment volumes based on volume priorities.

Another placement issue arises when a brick has many disks with different performance, such as fast, small, expensive disks and slow, large, cheap disks. Then, more commonly-used volumes could employ the faster disks, while seldom-used volumes employ the slower disks. Olive can support that, because the logical-to-physical mappings allow different physical disks to be used for dif-

ferent parts of a logical volume. For example, a volume may be placed a fast disk, while its snapshots are placed partly on the fast disk (for blocks shared with the source volume) and partly on the slow disk (for blocks that have diverged from the source volume). These techniques are not specific to distributed storage; existing solutions for centralized storage can be applied at each brick of Olive. Currently, Olive does not support splitting the sharing of storage across bricks—in other words, copy-on-write or move-on-write to a remote brick.

8 Evaluation

We implemented Olive within the Federated Array of Bricks (FAB), a distributed storage system [3, 20] that provides block-level storage with reliability and availability comparable to high-end disk arrays but without their high price tags. Bricks are built from commodity off-the-shelf hardware, and they provide an iSCSI interface to storage. For fault tolerance, FAB uses quorum-based data replication, as explained in Section 6.1. We modified FAB to keep track of a version tree (Section 3) and implement the scheme described in Sections 6.2–6.8. In FAB, the set of contiguous blocks stored at a replica group is called a *segment*, typically of size 1 GB. The *segment map* indicates for each segment which replica group stores it.

Olive inherits good availability and I/O performance from FAB, described in [3, 20], and so our evaluation of Olive focuses on branching and how it affects the system. The metrics we consider are the following:

- Branch latency: the time taken to create a clone or snapshot. This metric is important to users of the system, who (presumably) want these to be created "as soon as possible".
- I/O delay while branching: the length of time I/O's
 are delayed while a snapshot or clone is being created. This metric affects overall throughput and
 helps define acceptable frequencies with which storage branches can be made. This and the branch latency are key metrics for evaluating the general performance of branching storage.
- Metadata size: the size of the sharing lists, logicalto-physical maps, and segment maps required for each storage branch. This information is needed for each storage branch, and should not be too large.
- I/O latency for a branched volume: the latency for I/O after a branch has been created. One expects somewhat higher latency in this case for the first write to each block, which requires splitting the sharing by doing copy-on-write or move-on-write. This and the metadata size results are more specific to FAB than Olive, although they help illustrate some key overheads in the overall performance of the system.

Acceptable values for the above metrics vary per application, and since storage is intended to be a general-purpose service, the better the numbers, the more useful the service is.

8.1 System configuration

We use rack-mounted servers as bricks, each with two 1GHz Pentium 3 CPUs,³ 2GB of memory, three Seagate Cheetah 32GB SCSI disks (15K rpm, 3.6ms average seek time), and two Intel Gigabit Ethernet interfaces. They run Debian 3.0 with the Linux 2.6.9 kernel. The first 6GB of one disk hosts Linux and FAB/Olive software, while the remaining 90GB is used for FAB/Olive storage. Up to 20 machines are bricks, and other machines generate workload as needed.

In all experiments, we use 3-way replication and ensure that segments and data are distributed evenly to provide similar load to bricks.

For some experiments, we used an alternative branching algorithm from the original version of FAB. This algorithm uses Paxos to distribute a "create clone" message to all bricks, which then update their global volume metadata. This algorithm does not delay I/O's or synchronize them with the clone operation, which may cause bricks to process I/O and the clone creation in different orders. We refer to this algorithm as the *straw man algorithm*.

8.2 Metadata size

Metadata for branching storage includes the logical-to-physical maps and the sharing lists, kept in memory. There are 8 bytes for each entry in the segment map and 12 bytes for each entry in the sharing list. Figure 10 shows metadata size per brick per branch, for small, medium, and large volumes stored on 8 or 16 bricks, using either 512 KB or 1 MB disk allocation sizes (see Section 5), and a segment size of 1 GB. Metadata size doubles when the number of bricks is halved because each brick stores a larger part of the storage volume.

From these numbers, with a 1 MB disk allocation size, 16 bricks and 200 MB of memory in each brick for metadata, a large volume can have over 50 branches, while a medium volume can have over 400 branches. While we keep metadata in main memory, it is possible to page this information, allowing for a virtually unlimited number of branches.

8.3 Branch latency

There are two major components to the user-visible time for creating branches (clones or snapshots). The first component is the time for uniform confirmed broadcast, which we expect to depend mostly on the number of bricks. The second is the time for each brick to copy/create/modify volume metadata; this time is bro-

Volume size	# bricks	Metadata size (das=512 KB)	Metadata size (das=1024 KB)
24 GB	8	216 KB	108 KB
24 GB	16	108 KB	54 KB
192 GB	8	1730 KB	866 KB
192 GB	16	866 KB	434 KB
1536 GB	8	13836 KB	6924 KB
1536 GB	16	6924 KB	3468 KB

Figure 10: Amount of metadata per brick per branch for a small, medium, and large volume in a system with 1 GB segment size, 8 or 16 bricks, using 512 KB or 1024 KB disk allocation sizes (das).

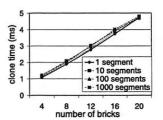


Figure 11: Clone creation latency as a function of number of bricks and segments. The segment size was fixed, and volume size chosen to result in the desired number of segments. Error bars show 95% confidence intervals.

ken up as two subcomponents: handling volume metadata (segment maps) and handling disk metadata (sharing lists). The first subcomponent should vary only with the number of segments in the volume, while the second subcomponent should depend mostly on whether the storage location has ever been written (if not, physical disk space will not have been allocated, and there will be no data to copy), and the amount of physical storage allocated on each brick. Neither subcomponent should depend on the number of existing branches of a volume.

Figure 11 shows the latency for clone creation versus the numbers of bricks and segments in a volume. The volume had no back-end physical storage allocated, which eliminates disk metadata copying. The number of segments varies from 1 to 1024, which with 1 GB segments represents a 1 TB volume. From the figure, we see that latency varies negligibly with number of segments, which indicates that snapshot latency depends primarily on the time for broadcast, not on segment handling. Indeed, separate measurements show that segment handling time is less than 30 μs in all experiments. The latency increases linearly with the number of bricks, indicating that the bottleneck is the overhead of sending and receiving uniform confirmed broadcast messages (the number of messages for this operation is also linear in the number of bricks). We conclude that optimizing the broadcast mechanism could significantly improve clone latency, if necessary.

We repeated these experiments using the straw man algorithm and got far worse results (approximately 50 ms higher latency in all cases), which indicates our custom

³Only one CPU per brick is actively used during the evaluation, because FAB is single-threaded.

Disk allocation size	Disk allocation units per brick	Metadata size per brick
128 KB	73 728	864 KB
512 KB	18 432	216 KB
1024 KB	9216	108 KB
4096 KB	2 304	27 KB

Figure 12: Size of metadata for a small 24 GB volume with 3-way replication as we vary the disk allocation size, for a system with 8 bricks and segment size of 1 GB. These numbers are identical for a 48 GB volume and 16 bricks.

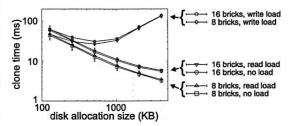


Figure 13: Clone creation latency while varying the size of disk allocation, and type of I/O workload (either none, or 100% read or write) on the system. Error bars show 95% confidence intervals

protocol provides not only better consistency but also better performance.

We also conducted experiments to measure how disk metadata size affects branching latencies. Since the disk metadata size is dominated by the number of disk allocation units, we varied the disk allocation size from 128 KB to 4096 KB while holding the volume and segment size constant, to produce varying amounts of metadata.

For small volumes with 24 GB or 48 GB, the parameters used are shown in Figure 12, and results are shown in Figure 13 (the curves labeled "no load"). It can be seen that disk metadata handling contributes significantly to latency. The copying of metadata starts during phase 2 of the broadcast, and theoretically proceeds in a separate thread independently of future broadcast phases. In practice, since FAB uses only one CPU and non-preemptive threads, the metadata copying thread typically must finish before processing of later broadcast phases, which adds to clone latency. Moreover, when copying the metadata, all of the volume's metadata is locked, preventing I/O's from completing. We conclude that one could reduce clone time and I/O latencies during clone by using multiple processors and allowing for incremental metadata copying.

For a medium volume of size 192 GB (8 bricks) or 384 GB (16 bricks), the latencies are ≈8 times larger than for the small volume size. This suggests using correspondingly larger disk allocation sizes to obtain the same performance.

Figure 13 also shows clone latency when the system is under load; we used a random workload with 1 KB

reads or writes. For the read workload, the clone latency rises slightly because processing of clone requests compete with ongoing I/O. For the write workload, the clone latency rises much more with increasing disk allocation size, because of copy-on-write: each 1KB write results in 3 times the disk allocation size of data movement. This I/O and memory activity delays the handling of the broadcast messages of clone operations, increasing the clone times, although they are still well within acceptable margins. This information can be used to fine-tune the desirable range for disk allocation unit size. 512KB to 1MB yields efficient clone operations, while providing a good trade-off between efficiency, locality and metadata size.

8.4 I/O delay while branching

This metric is the length of time that I/O's are delayed while branching executes. In Olive, branching may cause a coordinator to retry the first phase of the I/O protocol while a volume version is not stable (Section 6.5).

Thus, the I/O delay while branching is $T_0 + T_1$, where T_0 is the time it takes between phases 2 and 3 of uniform confirmed broadcast (which is when a version number is not stable), and T_1 is the time it takes to retry the first phase of the protocol. T_0 is about 1/4 of the total time to create a new branch (see Section 8.3) and should not be more than a few tens of milliseconds. T_1 is a fraction of a millisecond, and it is dominated by T_0 . We conclude that the total delay is about 1/4 of the time to create a new branch. Figure 14 shows the read latencies for a random workload while a snapshot is taken. The system had 16 bricks, and disk allocation size was 512KB. Data was in cache to avoid large variations from disk I/O and further accentuate the snapshot overheads. From Figure 13, the total snapshot time is ≈ 18 ms. The actual I/O delay is ≈4.8ms, as seen in the enlarged part of Figure 14. This confirms that I/O delays are equal to about 1/4 of the time to create a branch. Some I/O operations are delayed by up to 4ms, visible in the brief period immediately after the snapshot delay.

8.5 I/O latency and throughput for a branched volume

We now compare the latency of I/O between a nonbranched volume and a branched volume. A branched volume requires breaking data sharing the first time that a block is written, by using copy-on-write or move-onwrite.

For read operations, we expect the I/O latency and throughput to be unchanged, and this is shown in Figure 14. Figure 15 shows the result for write, where there is a visible latency penalty, resulting from copy-on-write, rather than snapshot overheads. Over time, as the underlying storage is increasingly separated, the throughput and latency return to normal, as seen by the time-average points in the figure.

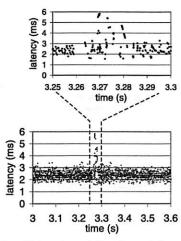


Figure 14: Read latencies while a branch is created. Each point shows the latency of one I/O. The period surrounding the snapshot is magnified, showing a gap while the branch is created at around time 3.26s.

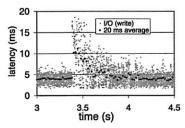


Figure 15: Write latencies while a branch is created. Grey points show the latency of one I/O; black points show the average latency in the previous 20ms.

8.6 Correctness

We did some regression tests to test branch consistency. The first test issues many pairs of synchronous writes while snapshots are taken, and then tests consistency by verifying that the snapshots do not incorporate the second write of a pair without the first. While Olive did not show any consistency errors, approximately 25% of the snapshots taken using the straw man solution did.

In the second test, we introduce partial writes to a volume to simulate coordinator failures, and then we take a snapshot and read the source and snapshot volumes while bricks fail and recover. These reads should generate a write back for either the old data (data before the partial write) or new data, depending on how it executes. Inconsistency occurs if a snapshot has the new data while its source has the old data, since the snapshot would represent a state that never happened in its source volume. While Olive carefully considers which versions are updated on a write back, the straw man solution simply writes back to the volume read. Our implementation did not show any inconsistencies, whereas the straw man did.

8.7 Evaluation summary

We have shown that the time to create a branch in Olive is primarily affected by the time taken for the broadcast, disk metadata manipulation and copy-on-writes. The most important parameter for our system was the disk allocation size, with larger sizes resulting in less metadata (and consequently faster branch operations) but more copy-on-write overhead (and slower branches). A size of 512KB seems ideal, resulting in snapshot times of a few tens of milliseconds in the worst case. In that case, the I/O delay while branching is also a few milliseconds. These latencies are on the same order as the I/O latencies themselves, and smaller than for those I/O's that must go to disk.

Better performance is possible by improving the time to do a copy-on-write or move-on-write locally on a brick, and increasing the concurrency within each brick, so that branch operations do not get queued behind I/O operations. However, this has nothing to do with the distributed nature of Olive, and involve existing techniques that are already in use in commercial products like disk arrays.

9 Related work

As we mentioned, Olive is built on top of FAB [3, 20], which provides distributed block storage, but without branching capabilities. For fault tolerance, FAB uses quorum-based data replication, as described in Section 6.1. There are many *centralized* or *single-server systems* that can capture consistent versions of data for backups or future perusal, including file systems (e.g., [19, 10, 21]), and database systems (e.g., [24, 18]). In all these systems, data is in a single place, so there are no issues of distribution and replication of data for consistency. State-of-the-art disk arrays support point-in-time branching (both snapshots and clones), and other forms of point-in-time copy. These systems are also centralized.

Petal [14] is a distributed replicated block storage system that supports (read-only) snapshots without consistency; intention to provide consistency has been announced [14], but no schemes have been proposed in the literature. Frangipani [25] is a distributed file system built on top of Petal; it provides (read-only) snapshots of file systems, by using the underlying snapshot facility of Petal. Consistency is obtained by pausing I/O at all nodes before taking a Petal snapshot, causing a potentially disruptive system hiccup. Moreover, if there are failures during the snapshot, nodes will be left in a paused state for even longer. Snapshots in Frangipani need not worry about distributed replicated data, since that is provided by Petal. Neither Petal nor Frangipani support clones.

Gifford was the first to propose replication of data at a majority of nodes; the work assumes some transactional support in the form of a stable file system [5]. Algorithms for quorum-based data replication over message-passing were proposed in the context of emulating shared mem-

ory using message-passing where some nodes may fail but the memory should retain its contents [2, 15, 16, 6]. Emulating shared memory means implementing primitives to read and write data, and so these algorithms lead to data replication schemes. Algorithms for quorum-based data replication in the context of real storage systems have been proposed for FAB [3, 20]. None of these algorithms support branching. A snapshot scheme has been proposed for FAB [11], but it does not provide consistency.

The work that is closest to ours is the Timeline [17] system, which provides consistent (read-only) snapshots for Thor, a system where a set of servers provide persistent storage for objects, and clients access these objects using a transactional interface. In essence, Timeline gets snapshot consistency by using logical clocks [12] implemented in a way different than the usual, but similar to what is suggested by Welch [26]. There are four main differences between our work and Timeline. First, Timeline does not support quorum-based replication of data. Second, Timeline requires modifications to storage clients: they piggyback and propagate timestamps internal to Thor. Moreover, Timeline clients must use the Thor abstraction for object-based storage, which precludes the use of existing storage applications. Third, Timeline does not support clones. And lastly, Timeline provides a weaker consistency guarantee than linearizability, which leads to inconsistency if the application nodes communicate outside of Thor (e.g., by sending messages over the network). Another scheme for taking (read-only) snapshots in Thor is Snap [23], but there are no details of how to ensure consistency with multiple servers.

10 Conclusion

In this paper, we described Olive, a distributed storage system that provides point-in-time branching. With Olive, storage branches can be created efficiently while providing a strong form of consistency. Olive provides a block-level interface to storage and requires no changes to storage clients. Today, we expect branching to be triggered by operators for tasks like "what-if" testing or quick storage provisioning. In the future, branching could be triggered by management applications to control the behavior of other applications by forking or rolling back their state.

Acknowledgments We thank Minwen Ji, Arif Merchant, and Yasushi Saito for insightful discussions. We thank Jay Wylie, our shepherd Steven Hand, and the anonymous referees for suggestions that improved the paper.

References

- M. K. Aguilera and S. Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Laboratories, 2003.
- [2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in messagepassing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
- [3] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Proceedings of the In*ternational Conference on Dependable Systems and Networks (DSN 2004),

- pages 125-134, June 2004.
- [4] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.
- [5] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles (SOSP 1979)*, pages 150–162, December 1979.
- [6] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International* Conference on Dependable Systems and Networks (DSN 2003), pages 259– 268, June 2003.
- [7] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems.
 In S. J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145.
 Addison-Wesley, 1993.
- [8] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.
- [9] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463–492, July 1990.
- [10] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Confer*ence, pages 235–246, January 1994.
- [11] M. Ji. Instant snapshots in a federated array of bricks. Technical Report HPL-2005-15, HP Laboratories Palo Alto, January 2005.
- [12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-565, July 1978.
- [13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), September 1979.
- [14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1996), pages 84–92, October 1996.
- [15] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the* 27th International Symposium on Fault-Tolerant Computing (FTCS 1997), pages 272–281, June 1997.
- [16] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th Interna*tional Conference on Distributed Computing (DISC 2002), pages 173–190, October 2002.
- [17] C.-H. Moh and B. Liskov. TimeLine: A high performance archive for a distributed object store. In Proceedings of the First Symposium on Networked Systems Design an Implementation (NSDI 2004), pages 351–364, March 2004.
- [18] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. IEEE Transactions on Knowledge and Data Engineering, 7(4):513– 532. August 1995.
- [19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221– 254, Summer 1995.
- [20] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. In Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS 2004), pages 48– 58, October 2004.
- [21] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 110–123, December 1999.
- [22] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. RFC3720: Internet small computer systems interface (iSCSI). http://www.faqs.org/rfcs/rfc3720.html, 2004.
- [23] L. Shrira and H. Xu. Snap: Efficient snapshots for back-in-time execution. In Proceedings of the 21st International Conference on Data Engineering (ICDE 2005), pages 351–364, March 2004.
- [24] M. Stonebraker. The design of the POSTGRES storage system. In Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 1987), pages 289–300, September 1987.
- [25] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 224–237, October 1997.
- [26] J. L. Welch. Simulating synchronous processors. Information and Computation, 74(2):159–171, August 1987.

Pastwatch: a Distributed Version Control System

Alexander Yip, Benjie Chen and Robert Morris MIT Computer Science and AI Laboratory

yipal@mit.edu, benjie@csail.mit.edu, rtm@csail.mit.edu

Abstract

Pastwatch is a version control system that acts like a traditional client-server system when users are connected to the network; users can see each other's changes immediately after the changes are committed. When a user is not connected, Pastwatch also allows users to read revisions from the repository, commit new revisions and share modifications directly between users, all without access to the central repository. In contrast, most existing version control systems require connectivity to a centralized server in order to read or update the repository.

Each Pastwatch user's host keeps its own writable replica of the repository, including historical revisions. Users can synchronize their local replicas with each other or with one or more servers. Synchronization must handle inconsistency between replicas because users may commit concurrent and conflicting changes to their local replicas. Pastwatch represents its repository as a "revtree" data structure which tracks the relationships among these conflicting changes, including any reconciliation. The revtree also ensures that the replicas eventually converge to identical images after sufficient synchronization.

We have implemented Pastwatch and evaluate it in a setting distributed over North America. We have been using it actively for more than a year. We show that the system is scalable beyond 190 users per project and that commit and update operations only take 2-4 seconds. Currently, five users and six different projects regularly use the system; they find that the system is easy to use and that the system's replication has masked several network and storage failures.

1 Introduction

Many software development teams rely on a version control system (VCS) to manage concurrent editing of their project's source code. Existing tools like CVS[7] and

Subversion[22] use a client-server model, where a repository server stores a single master copy of the version history and the clients contact the server to read existing revisions and commit new modifications. This model works well when the users can contact the server, but as portable computers gain popularity, the client-server model becomes less attractive. Not only can network partitions and server failures block access to the repository, but two clients that cannot contact the server cannot share changes with each other even if they can communicate directly.

One approach to solving this problem is to optimistically replicate the repository on each team member's computer. This would allow users to both modify the replica when they are disconnected and to share changes with each other without any central server. The challenge in this approach is how to reconcile the write-write conflicts that occur when two users independently modify their replicas while disconnected. Conflicts can occur at two levels. First, the repository itself is a complex data structure that describes the revision history of a set of files; after synchronizing, the repository must contain all the concurrent modifications and the system's internal invariants must be maintained so that the VCS can still function. The second level is the source code itself which also contains interdependencies. The VCS should present the modification history as a linear sequence of changes when possible but if two writes conflict, the system should keep them separate until a user verifies that they do not break interdependencies in the source code.

Pastwatch is a VCS that optimistically replicates its repository on each team member's computer. To manage concurrent modifications, Pastwatch formats the repository history as a *revtree*. A revtree is a data structure that represents the repository as a set of immutable keyvalue pairs. Each revision has a unique key and the value of each pair represents one specific revision of all the source code files. Each revision also contains the key of the parent revision it was derived from. Each time a

user modifies the revtree, he adds a new revision to the revtree without altering the existing entries. Revtrees are suitable for optimistic replication because two independently modified replicas can always be synchronized by taking the union of all their key-value pairs. The resulting set of pairs is guaranteed to be a valid revtree that contains all the modifications from both replicas. If two users commit changes while one or both is disconnected, and then synchronize their replicas, the resulting revtree will represent the conflicting changes as a *fork*; two revisions will share the same parent. Pastwatch presents the fork to the users who examine the concurrent changes and explicitly reconcile them.

Although Pastwatch users can synchronize their replicas with each other directly, a more efficient way to distribute updates is for users to synchronize against a single rendezvous service. In a client-server VCS, the repository server functions as the rendezvous but it must enforce single copy consistency for the repository. The consistency requirement makes it challenging to maintain a hot spare of repository for fail-over because a server and a spare may not see the same updates. Revtrees, however, support optimistic replication of the repository, so Pastwatch can easily support backup rendezvous servers with imperfect synchronization between servers. Pastwatch exploits the revtree's tolerance for inconsistency and uses a public distributed hash table that makes no guarantees about data consistency as a rendezvous service.

This paper makes three contributions. First, it describes the revtree data structure which makes divergent replicas easy to synchronize. Second, it shows how revtrees can handle many classes of failure and present them all to the users as forks. Finally, it describes Pastwatch, a distributed version control system that uses a replicated revtree to provide availability despite system failures, network failures and disconnected users.

We have implemented Pastwatch and have been using it actively for more than a year. We show that the system scales beyond 190 members per project and that commit and update operations only take 2-4 seconds. Currently, five users and six projects use the system, including this research paper and the Pastwatch software itself. The system has performed without interruption during this time despite repeated down-time of rendezvous nodes. During the same time, our CVS server experienced three days with extended down-time.

The remainder of this paper is organized as follows: Section 2 motivates Pastwatch and gives concrete requirements for its design. Section 3 discusses revtrees and section 4 describes how Pastwatch presents optimistic replication to its users. Sections 5 and 6 describe implementation details and system performance. Section 7 describes related work and Section 8 concludes.

2 Design Requirements

The task of a VCS is to store historic revisions of a project's files and to help programmers share new changes with each other. Ideally, a VCS would be able to accomplish these goals despite network disconnections, network failures and server failures. We outline the requirements of such a VCS below.

Conventional Revision Control: Any VCS should provide conventional features like checking out an initial copy of the source code files, displaying differences between file revisions and committing new revisions to the repository. In most cases, users will expect the system to have a single latest copy of the source code files, so when possible the VCS should enforce a linear history of file modifications.

At times, one or more project members may choose to fork to keep their modifications separate from other users. A fork is a divergence in the change history where two different revisions are derived from the same parent revision. A branch is a sequence of changes from the root revision to one of the current leaf revisions. After a fork, each of the two branches will maintain a separate sequential history and they will not share changes until they are explicitly reconciled. Forking is a common practice in software projects; for example, many projects use a main development branch and fork at each major release to create a maintenance branch. Some projects even use separate branches for each individual bug fix. This way, a programmer can make intermediate commits for the bug fix in her own branch without interfering with other programmers.

Disconnected Repository Operations: A VCS should support as many functions as possible even if it is disconnected from the network, for example when a user is traveling. The ability to retrieve old revisions from a local replica of the repository while disconnected is useful and easy to support. Being able to commit new revisions to the repository while disconnected is also useful, because programmers often commit changes several times a day.

For example, we will show in Section 6.1.1 that the average developer in the Gaim open-source project commits an average of 3 times per day when he is active and on the busiest day, a single user made 33 commits. Frequent commits are encouraged in software projects like PHP and libtool; their coding standards encourage programmers to make several smaller commits rather than a single large commit because it simplifies debugging.

A VCS that allows disconnected commits must handle conflicting commits. When two disconnected users commit changes, they do so without knowledge of the

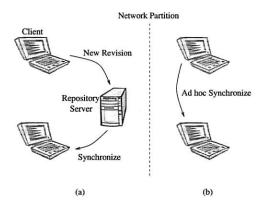


Figure 1: Flexible sharing of updates. Clients in one partition can share new revisions directly, without any servers.

other person's changes. This means that they may commit changes that conflict at a semantic level in the source code. In one example, two users may independently implement a sgrt function in a math library while disconnected. When they reconnect, the VCS could reconcile their changes automatically by including both sqrt implementations, but then the math library would fail to compile because the semantics of the resulting source code are invalid. Although concurrent modifications may not always conflict in this way, it is best to avoid situations where the repository revision does not compile or contains inconsistencies. Since it is difficult to automatically determine if concurrent changes will cause a source code inconsistency, the VCS should record the conflict and allow a human to make the distinction when convenient.

Flexible Update Sharing: A VCS should allow users to share their changes with each other whenever their computers can communicate. This includes scenarios where two users are sitting next to each other on an airplane; they are able to connect to each other but not to the VCS server or the other client hosts (see Figure 1b). They should be able to commit changes and share them with each other via the VCS, even though the repository server is not reachable.

Server Failures: Another scenario that the VCS should handle gracefully is a server failure. If a VCS server fails, the system should be able to switch to a backup server seamlessly. The event that motivated the Pastwatch project was a power failure in our laboratory one day before a conference submission deadline; the failure disabled our CVS server. We were able to create a new CVS repository off-site, but our history was unavailable and there was no simple way to reconcile the

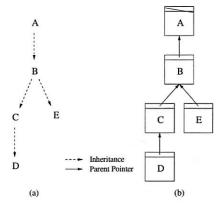


Figure 2: (a) Inheritance graph. Letters depict revisions. (b) Example revtree. Each box is a revision; it specifies one version of every file in the project. Nodes D and E are leaves and node B is a fork point.

change history between the old and new repositories after the power was restored. Ideally, the VCS would be able to manage update consistency between replicas so that switching between repository replicas would be easy.

3 Revtrees

Pastwatch supports disconnected operation by storing a full repository replica on each member's computer. If two users concurrently modify their repository replicas, there is a risk that the modifications will conflict when the users attempt to synchronize their replicas. Pastwatch ensures that neither modification is lost and that all replicas eventually reflect both modifications. That is, Pastwatch applies an optimistic replication strategy [25] to the repository.

Construction: The fundamental task of a repository is to store past revisions of the project files. Pastwatch stores these revisions in a *revtree* data structure that exploits the inheritance between immutable revisions to help it provide optimistic replication.

Each revision logically contains *one* version of *every* file in the project. Revisions are related through inheritance: normally a project member starts with an existing revision, edits some of the files, and then commits a new revision to the repository. This means each revision except the first one is a descendant of an earlier revision. Figure 2a illustrates this inheritance relationship between revisions A through E. The dashed arrow from A to B indicates that a user modified some files from revision A to produce revision B.

Pastwatch stores the repository as a revtree modeled after the inheritance graph. A revtree is a directed acyclic

graph, where each node contains a revision. Each revision is immutable and has a unique revision identifier called an RID. Each revision contains a *parent* pointer: the RID of the revision from which it was derived (see Figure 2b).

When a user commits a change to the repository, Pastwatch creates a new revision, adds it to the revtree in the user's local repository replica and finally synchronizes with the other replicas to share the new revision. If users commit new revisions one at a time, each based on the latest revision acquired via synchronization, then the revtree will be a linear revision history.

Handling Network Partitions: Users may not be able to synchronize their replicas due to lack of network connectivity. They may still commit new revisions, but these revisions will often not be derived from the globally most recent revision. These concurrent updates pose two problems: the overall revision history will no longer be linear, and the various repository replicas will diverge in a way that leaves no single most up-to-date replica.

When two users with divergent repositories finally synchronize, Pastwatch must reconcile their differences. Its goal is to produce a new revtree that reflects all changes in both users' revtrees, and to ensure that, after sufficient pair-wise synchronizations, all replicas end up identical. Each repository can be viewed as a set of revisions, each named by an RID. Revisions are immutable, so two divergent revtrees can only differ in new revisions. This rule holds even for new revisions that share the same parent, since the parent revision is not modified when a new child is added. Pastwatch chooses RIDs that are guaranteed to be globally unique, so parent references cannot be ambiguous and two copies of the same revision will always have the same RID no matter how many times the replicas are synchronized. These properties allow Pastwatch to synchronize two revtrees simply by forming the union of their revisions. Any synchronization topology, as long as it connects all users, will eventually result in identical repository replicas.

Revtrees gain several advantages by using the union operation to synchronize replicas. First, partially damaged revtrees can be synchronized to reconstruct a valid and complete replica. Second, the synchronization process can be interrupted and restarted without harming the revtrees. Finally, the synchronization system does not need internal knowledge of the revtree data structure; Section 5.2 describes how Pastwatch uses this property to store a replica in a distributed hash table.

Managing Forks: The usual result of commits while disconnected is that multiple users create revisions with the same parent revision. After synchronization, users will see a *fork* in the revtree: a non-linear revision history

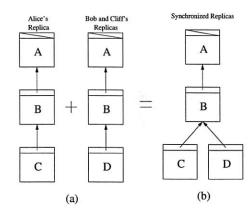


Figure 3: Forking Examples. (a) Two divergent revtree replicas. (b) The two divergent replicas from (a) are synchronized and the resulting revtree contains a fork.

in which one revision has multiple successors. Figure 3 illustrates the formation of a fork, caused by two disconnected users both creating revisions (C and D) based on revision B. Now the revtree has two leaves; the path from each leaf to the root node is called a *branch*.

A fork correctly reflects the existence of potentially incompatible updates to the project files, which can only be be resolved by user intervention. If nothing is done, the repository will remain forked, and users will have to decide which branch they wish to follow. This may be appropriate if the concurrent updates reflect some deeper divergence in the evolution of the project. However, it will often be the case that the users will wish to return to a state in which there is a single most recent revision. To reconcile two branches, a user creates a new revision, with the help of Pastwatch, that incorporates the changes in both branches and contains two parent pointers, referring to each of the two branch leaves. Ideally, the user should reconcile when he is connected to the network so that the reconcile is available to other users immediately; this avoids having other users repeat the reconcile unnecessarily. Figure 4a illustrates two branches, C and D, that are reconciled by revision E.

As with any commit, a disconnected user may commit a new child to revision C before he sees E. The resulting revtree is illustrated in Figure 4b. Once again, the revtree has two leaves: F and E. To reconcile these two branches a user proceeds as before. He commits a new revision G with parents E and F. The final branch tree is shown in Figure 4c. Two members can also reconcile the same two branches concurrently, but this is unlikely because Pastwatch will detect a fork when the diverging replicas first synchronize and suggest that the user reconcile it immediately.

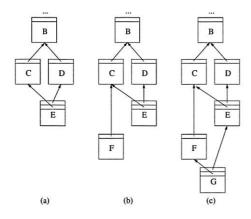


Figure 4: Reconciling Branches. (a) Revision E reconciles the fork and joins C and D. (b) Revision F creates a new fork and reuses C. (c) Revision G reconciles the new fork.

Synchronization Patterns: Pastwatch users may synchronize pairs of replicas in whatever patterns they prefer. One reasonable pattern is to mimic a centralized system: for every replica to synchronize against the same designated "rendezvous" replica. This pattern makes it easy for all users to keep up to date with the latest generally-available revision. Another pattern is adhoc synchronization which helps when users are isolated from the Internet but can talk to each other. Figure 5 illustrates both rendezvous and ad-hoc synchronization.

Revtree Benefits: Revtrees provide a number of key benefits to Pastwatch. First, revtrees provide flexibility in creating and maintaining replicas because they guarantee that the replicas will converge to be identical. For example, if a project's rendezvous service is not reliable, its users can fall back to ad-hoc mode. Alternatively, the users could also start or find a replacement rendezvous service and synchronize one of the user's local replicas with it, immediately producing a new working rendezvous replica.

Revtrees also aid with data corruption and data transfer. If two replicas are missing a disjoint set of revisions, they can synchronize with each other to produce a complete replica. Also, the new revisions are always easy to identify in a revtree, so synchronization uses very little bandwidth.

Revtrees handle several types of failure, using the fork mechanism for all of them. For example, if a rendezvous loses a leaf revision due to a disk failure, then another user could inadvertently commit without seeing the lost revision. After repairing the rendezvous, the visible evidence of the failure would be an implicit fork. Similarly, network partitions and network failures can result

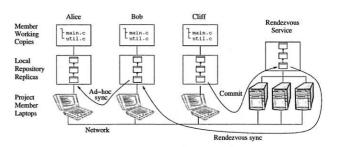


Figure 5: User-visible Model. Local repository replicas, rendezvous replica and working copies.

in forks. The revtree's eventual consistency ensures that the only impact of such failures is a fork. Users only need to learn one technique (reconciling forks) in order to deal with a wide range of underlying problems and as we show in Section 4, reconciling forks is not difficult, so using forks to handle failures is convenient for users.

4 User-Visible Semantics

This section explains the user's view of how Pastwatch works. To the extent possible, Pastwatch's behavior mimics that of CVS.

Working Copy: A Pastwatch user edits a working copy of the project files, stored as ordinary files in the user's directory. A user creates a working copy by checking out a base revision from the repository. The checkout command copies the files from the base revision into the working copy and remembers the working copy's base revision.

Tracking New Revisions: In order to see other users' new revisions, a user will periodically *update* her working copy. The update command first fetches new revisions from the rendezvous replica. It then checks if the working copy's base revision has any new children. If the base revision has just one child, Pastwatch will apply changes from the child to the working directory. Pastwatch will follow single children, merging them into the working directory with 3-way diff[6], until it reaches a revision with either zero or more than one child. Pastwatch changes the working directory's record of the base revision to reflect this last revision.

Committing New Revisions: In most cases, a linear history of changes is desirable, so Pastwatch will not create a fork if it can avoid it. When a user tries to commit new changes stored in the working copy, Pastwatch first tries to synchronize the local revtree against the rendezvous. It then checks whether the working copy's base

revision has any descendants. If the base revision does have new descendants, Pastwatch will refuse to create a new revision until the user updates his working copy.

There is a potential race between reading the base revision and appending a new revision. As an optimization, Pastwatch uses a best-effort leasing scheme to prevent this race from causing unnecessary forks. Pastwatch tries to acquire a lease on the repository before fetching the base revision and releases it after synchronizing the new revision with the rendezvous replica. When the user can contact the rendezvous service, Pastwatch uses the rendezvous service to store the lease. The lease is only an optimization. If Pastwatch did not implement the lease, the worst case outcome is an unnecessary fork when two connected users commit at exactly the same time. If the rendezvous is unavailable, Pastwatch proceeds without a lease.

Implicit Forks: If two disconnected users independently commit new revisions, an implicit fork will appear when synchronization first brings their revisions together. A user will typically encounter an unreconciled fork when updating her working copy. If there is an unreconciled fork below the user's base revision, Pastwatch warns the user and asks her to specify which of the fork's branches to follow. Pastwatch allows the user to continue working along one branch and does not force her to resolve the fork. This allows project members to continue working without interruption until someone reconciles the fork.

Explicit Forks: Pastwatch users can fork explicitly to create a new branch so that they can keep their changes separate from other members of the project. To explicitly fork, a user commits a new revision in the revtree with an explicit branch tag. Pastwatch ignores any explicitly tagged revisions when other users update.

Reconciling Forks: Both implicit and explicit branches can be reconciled in the same way. Reconciling forks is no more difficult than updating and committing in CVS. Figures 6 and 7 illustrate the process.

Forks first appear after two divergent replicas synchronize. In the examples, Alice synchronizes her local replica during an update and Pastwatch reports a new fork because both Alice and Bob made changes while Alice was disconnected from the network. To reconcile the fork, Alice first issues a reconcile command which applies the changes from Bob's branch into Alice's working copy.

In Figure 6, there were no textual conflicts while applying Bob's changes to Alice's working copy, so Alice

```
alice% past update
  Tracking branch: init, alice:3
 Branch "init" has forked.
 current branches are:
    branch "init": head is alice:3
    branch "init": head is bob:2
alice% past -i reconcile -t bob:2
  Tracking branch: init, alice:3
  updating
  Reconciling main.c
 M main.c: different from alice:3
alice% past -i -k bob:2 commit -m "Reconcile branches"
  Tracking branch: init, alice:3
  checking for updates and conflicts
  M main.c
  committing in .
  committing main.c
  Built snapshot for revision: alice: 4
```

Figure 6: Reconciling a fork without source code conflicts.

can just commit a new revision that is a child of both Alice's and Bob's revisions as shown in Figure 4a. In contrast, Figure 7 shows what Alice must do if the fork created a source code conflict. Pastwatch notifies Alice during the reconcile and inserts both conflicting lines into her working copy the way CVS reports conflicts during an update. After Alice resolves the conflict she can commit the final revision.

5 Implementation

The Pastwatch software is written in C++ and runs on Linux, FreeBSD and MacOS X. It uses the SFS tool-kit[18] for event-driven programming and RPC libraries. It uses the GNU diff and patch libraries to compare different revisions of a file and perform three-way reconciliation. Pastwatch is available at: http://pdos.csail.mit.edu/pastwatch.

5.1 Storage Formats

Pastwatch stores the entire local replica in a key-value store implemented by a BerkeleyDB database for convenience. All the replica data structures are composed of key-value pairs or *blocks*. Immutable blocks are keyed by the SHA-1[11] hash of their content.

For the sake of storage and communication efficiency, each revision in the revtree only contains the difference from the parent revision rather than an entire copy of the source code files. The internal representation of a revision is a combination of a *revision record* and *delta blocks*, all of which are immutable blocks. Delta blocks contain the changes made to the parent revision in the GNU diff format. Figure 8 illustrates the structure of a revision record. The RID of a revision equals the SHA-1

```
alice% past update
  Tracking branch: init, alice:3
 Branch "init" has forked.
  current branches are:
    branch "init": head is alice:3
    branch "init": head is bob:2
alice% past -i reconcile -t bob:2
  Tracking branch: init, alice:3
  updating .
  Reconciling main.c
  C main.c: conflicts with alice:3
alice% grep -A4 "<<<" main.c
  <<<<<< alice:3
     int increase (int x) { return x + 1; }
      void increase (int &x) { x++; }
  >>>>> bob:2
< Alice reconciles conflicting edits with a text editor >
alice% past -i -k bob:2 commit -m "Reconcile branches"
  Tracking branch: init, alice:3
  checking for updates and conflicts
  updating .
  M main.c
  committing in .
  committing main.c
  Built snapshot for revision: alice: 4
```

Figure 7: Reconciling a fork with a source code conflict.

hash of the revision record block. parent contains the RID of the parent revision. previous contains the key of the previous entry in the member log described in Section 5.2. The remainder of the revision record contains references to delta blocks. The revision record includes the first few delta blocks; if there are more deltas, Pastwatch will use single and double indirect blocks to reference the deltas. The arrangement of delta blocks was inspired by the UNIX file system's[19] handling of file blocks.

Pastwatch keeps a local snapshot of each revision's files and directories so that it can retrieve old revisions quickly. Pastwatch saves the snapshots locally in a CFS[9] like file system that reuses unchanged blocks to conserve storage space. Since revision records only contain deltas, Pastwatch constructs the snapshots by applying deltas starting at the root of the revtree. Since Pastwatch keeps all the snapshots, it only needs to construct snapshots incrementally when it retrieves new revisions. Snapshots are stored in the local key-value store.

5.2 Rendezvous Services

We have implemented two different rendezvous services for Pastwatch. First, we implemented a single server rendezvous service where all users synchronize their replicas with the single server. This service is fully functional, but if the server becomes unavailable, the users will probably need to use ad hoc synchronization to share changes which can be slow to propagate new changes. It is possi-

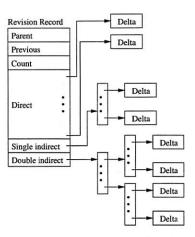


Figure 8: Revision record data structure with delta blocks.

ble to implement a hot spare replica for the single server but instead, we constructed a resilient rendezvous service using a distributed hash table(DHT)[27][10][17][33].

DHTs promise abundant, reliable storage and the arrival of public storage DHTs like OpenDHT[13][26] make them an attractive choice for a Pastwatch rendezvous service. Many different projects can all share the same DHT as their rendezvous service and since DHTs are highly scalable, one could build a large repository hosting service like Sourceforge[5] based on a DHT.

Revtrees are compatible with DHTs because a DHT is a key-value storage service and revtrees can tolerate the imperfect consistency guarantees of DHT storage. As shown in Section 3, revtrees handle network partitions, missing blocks and slow update propagation, so a storage inconsistency in a rendezvous DHT will at worst cause a fork in the revtree. The only additional requirement of the DHT is that it must support mutable data blocks so that Pastwatch can discover new revisions.

Pastwatch uses mutable blocks and one extra data structure when using a DHT in order to discover new revisions; this is because the put/get DHT interface requires a client to present a key to get the corresponding data block. Each revtree arc point upwards, towards a revision's parent; the revtree does not contain pointers to the newest revisions, so Pastwatch must provide a way to discover the keys for new revisions. Pastwatch accomplishes this by storing the revisions in a per-user log structure that coexists with the revtree; the structure is rooted by a mutable DHT block. The address of the mutable block is an unchanging repository ID. Pastwatch can find the new revisions as long as it has the repository ID, thus it can find all revisions in the revtree.

Figure 9 illustrates the revtree DHT structures. In this example, the project has two developers, Alice and Bob.

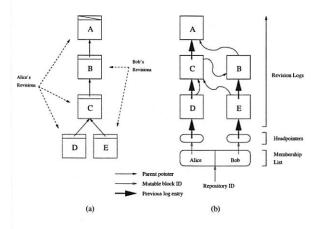


Figure 9: DHT storage structures. (a) shows an example revtree. (b) shows the same revtree, as stored in the DHT. Rounded rectangles are mutable blocks and square rectangles are immutable blocks.

Figure 9a shows the repository's revtree including who created each revision. Figure 9b shows how the revtree is stored in the DHT.

Each user has their own log that contains every revision they created. Each user maintains a pointer to their newest revision in a mutable block called a *headpointer*. The project's membership list contains a pointer to each of the users' headpointers. All the blocks in the DHT structure are immutable except the headpointers and the membership list.

It is efficient to synchronize replicas because finding the newest revisions is efficient. Pastwatch only needs to scan the membership list and traverse each user's log until it encounters an RID it has encountered in the past because the newest revisions are at the front of the log. Since the revisions are immutable, Pastwatch can be sure that the remainder of the log has been processed at an earlier time.

5.3 DHT Implementation

At the time of writing, OpenDHT is the only public DHT storage service we are aware of that implements the put/get interface. Although OpenDHT provides the correct interface, it will purge a data block after storing it for a week unless the block is inserted again. Pastwatch cannot use OpenDHT because Pastwatch reads old blocks during a fresh checkout and a checkout from the DHT replica will fail if a block is unavailable. We implemented our own DHT that provides long-term storage, but Pastwatch can be modified to use a suitable public storage DHT if one becomes available.

The Pastwatch DHT rendezvous service is derived from Dhash[9][10]. Immutable blocks are stored under

the SHA-1 hash of their content. Each mutable block has a single writer and the DHT only allows mutations that are signed by a private key owned by the writer. Each mutable block has a constant identifier equal to the hash of the owner's public key. Each mutable block contains a version number and the owner's public key along with the block's payload. Each time an owner updates his mutable block, he increases the version number and signs the block with his private key. The DHT stores the block along with the signature and will only overwrite an existing mutable block if the new block's version number is higher than the existing block and the signature is correct.

5.4 Data Durability

The Pastwatch DHT uses the IDA coding algorithm [23] to provide data durability. For each block, the DHT stores 5 fragments on different physical nodes and requires 2 fragments to reconstruct the block. The DHT also actively re-replicates blocks if 2 of the fragments become unavailable. Data loss is unlikely because the nodes are well maintained server machines, but if the DHT does experience a catastrophic, corollated failure. any user with an up-to-date local replica can perform a repair by synchronizing his local replica with the rendezvous service. Alternatively, he could easily create a new single server rendezvous service. In either case, synchronizing his local replica will completely repopulate the empty rendezvous service. A corrupt replica on the rendezvous services can also be repaired by synchronizing with a valid local replica and in some cases, two corrupt replicas can repair each other simply by synchronizing with each other.

In practice, each Pastwatch project must evaluate its own data durability requirements. If a project has many active members who keep their local replicas up-to-date, then the members may elect to forgo any additional backup strategy. On the other hand, a project with only one member may choose to keep regular backups of the member's local replica.

6 Evaluation

This section evaluates the usability and performance of Pastwatch. First, we analyze a number of open-source projects and find that real users frequently commit 5 or more times a day, enough that they would want disconnected commits during a long plane flight. We also find that in a real 26-person team, 5 or fewer team members commit in the same day 97% of the time which suggests that even a day-long network partition will not overwhelm a Pastwatch project with implicit forks. We

then share experiences from a small initial user community which has been using Pastwatch for more than a year. In that time, Pastwatch has been easy to use and survived a number of network and storage failures. In the same time period our CVS server experienced significant down-time.

We then show that Pastwatch has reasonable performance. Common operations in our experimental workload, like commit, take 1.1 seconds with CVS and 3.6 seconds with Pastwatch. Pastwatch can also support many members per project; increasing the number of members from 2 to 200 increases the update time from 2.3 seconds to 4.2 seconds on a wide-area network. We also show that retrieving many old revisions is not expensive; pulling 40 new revisions from the rendezvous replica and processing the revisions locally takes less than 11 seconds.

6.1 Usability Evaluation

6.1.1 Disconnected Operations:

To evaluate the usefulness of the ability to commit while disconnected, we analyze the per-member commit frequency of real open-source projects. We find that it is common for a single project member to commit several new revisions in a single day and conclude that the ability to commit while disconnected more than a few hours would be useful.

We analyzed the CVS commit history from three of the more active open source projects hosted on the Sourceforge[5] repository service: Gaim, Mailman and Gallery. Figure 10 characterizes the daily commit activity for all members in each project for days that contain commits. The plot shows that the median number of commits is relatively low at only 2 commits, but there is a significant fraction of days in which a single user commits 5 or more times. In 18% of the active days, a single Gallery member made 5 or more commits in a single day. In 22% of the active days, a single Mailman member made 7 or more commits in a single day.

Considering that most users will be programming fewer than 16 hours in a day, the high daily commit counts suggest that even a disconnection period of 3-5 hours would interrupt a user's normal work-flow and so disconnected commits could be useful for these projects.

6.1.2 Commit Concurrency:

Pastwatch users are able to commit while disconnected or partitioned so there is a risk that many project members will commit concurrently and create a large number of implicit forks. To evaluate how often disconnected commits would actually result in an implicit fork, we

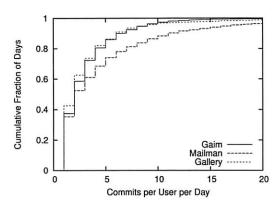


Figure 10: Cumulative distribution of per user, daily commit counts. In 18% of the active days, a single Gallery member made 5 or more commits in a single day. In 22% of the active days, a single Mailman members made 7 or more commits in a single day.

analyzed the temporal proximity of commits by different project members in real open-source projects. We found that different project members do commit at similar times, but the level of concurrency should not cause a large number of forks.

The number of forks that may result from a network partition is limited to the number of partitions because replicas in the same partition can always synchronize with each other and they should not accidentally create a fork within their partition. The worst case occurs when every member in a project commits a new revision while they are all partitioned from each other. This scenario results in a separate branch for each member. To evaluate the likelihood of the worst case forking scenario, we analyzed the CVS logs for the same three open-source projects used in Section 6.1.1.

The Gaim, Mailman and Gallery projects have 31, 26 and 21 active members respectively, so the worst case number of branches is quite high. The highest number of unique committers in a single day, however, was only 9, 6 and 5 respectively. Even if all the members in each project were partitioned into individual partitions for a 24 hour period and they made the same commits they made while connected, the number of resulting forks in each project would still be quite low and significantly fewer than the total number of members in the project.

The low number of concurrent commits on the highest concurrency day already suggests that the number of implicit forks will be manageable, but to better understand the common case, we consider the distribution of unique committers. Figure 11 shows the distribution of the number of unique users who commit in a calendar day. Mailman sees three or fewer unique committers 99% of the time and Gaim sees five or fewer unique com-

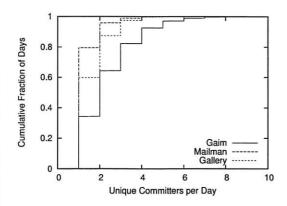


Figure 11: Cumulative distribution of unique committers per day. Mailman sees three or fewer unique committers 99% of the time and Gaim sees five or fewer unique committers 97% of the time.

mitters 97% of the time.

The distribution suggests that the number of concurrent committers is normally low with respect to the number of project members. The low frequency of concurrent commits combined with the ease of reconciling forks described in Figure 6 suggests that implicit forks will be manageable in practice.

6.1.3 Practical Experience:

Pastwatch currently has a user community of five people and six projects: three documents and three software projects including this research paper and the Pastwatch software itself. All Pastwatch users agree that the system is as usable and convenient as CVS.

The Pastwatch users primarily use connected mode and the system has behaved like a traditional centralized VCS. On occasion, the users also use disconnected reads and commits. For example, this paper's repository has been active for 202 days. During that time, it has served 816 repository operations including updates, commits, checkouts and diffs; 25 of those operations were performed while the client was disconnected from the network. Out of the 816 operations, there were 181 commits; seven of those commits were performed while the client was disconnected from the network.

All Pastwatch projects use the Pastwatch DHT as their rendezvous service, and it has proven to be robust. In the past year, our research group's main file server experienced three days with extended down-time. Since many people store their CVS repositories on the file server, they could not commit changes or read old revisions from the repository during the down-time. Pastwatch users were able to read and write to their local replicas while the file server was unavailable.

6.2 Performance Evaluation

6.2.1 Experiment Setup:

The following experiments though Section 6.2.5 are based on the following setup: CVS is configured with a single CVS server in Cambridge, Massachusetts. It has two different client hosts; one is in New York, New York and the other is in Salt Lake City, Utah. The client in New York has a 1.1 GHz CPU and a 4.3 MB/s bidirectional bottleneck bandwidth to the CVS server with a 6ms round trip latency. The host in Utah has a 1.7 GHz CPU and 0.5 MB/s bottleneck bandwidth to the CVS server and a 55ms round trip latency.

Pastwatch uses the same two client hosts and an 8 node DHT. The client host in New York accesses the DHT through a node with a 6ms round trip latency. The client host in Utah connects to a nearby DHT node with a 13ms round trip latency. Four of the DHT nodes are spread over North America and the other four are located in Cambridge. The New York client and many of the DHT nodes are on the Internet2 research network but the Utah client is not, so the New York client has higher throughput links to the DHT than the Utah client.

The base workload for each experiment is a trace from the CVS log of the SFS open-source software project. The trace begins with 681 files and directories and includes 40 commit operations. On average, each commit changes 4.8 files, the median is 3, and the highest is 51. Together, the 40 commit operations modify roughly 4330 lines in the source-code and add 6 new files. Each data point is the median of 10 trials and for each trial, Pastwatch used a different repository ID and different head-pointer blocks.

6.2.2 Basic Performance:

This section compares the performance of basic VCS operations like import, checkout, update and commit in Pastwatch and CVS. Their times are comparable, but round trip times and bottleneck bandwidths affect them differently.

In each experiment, the primary client host creates the project and imports the project files. Each client then checks out the project. Afterwards, the primary client performs the 40 commits. After each commit, the secondary client updates its replica and working copy to retrieve the new changes. The experiment was run once with the New York client as primary and again with the Utah client as primary. The Pastwatch project has two members.

Table 1 reports the costs (in seconds) of the import operation, the checkout operation, and the average costs of the commit and update operations for each client running the workload.

	New York Client				Utah Client			
	import	checkout	mean commit	mean update	import	checkout	mean commit	mean update
CVS	5.4	5.8	1.1	2.9	13.0	10.5	2.2	3.8
Pastwatch	167.4	16.3	3.6	3.0	161.4	25.9	3.9	2.4

Table 1: Runtime, in seconds, of Pastwatch and CVS import, checkout, commit, and update commands. Each value is the median of running the workload 10 times. The update and commit times are the median over 10 trials of the mean time for the 40 operations in each workload.

Since Pastwatch creates a local repository replica during import and checking out a working copy from a complete replica is trivial, the checkout time for the client that imported is not reported here. Instead, we report the checkout time on the client that did not import.

Initially importing a large project into CVS takes much less time than with Pastwatch because CVS stores a single copy of the data while the Pastwatch DHT replicates each data block on 5 different DHT nodes. In practice, a project is only imported once, so import performance is not very significant.

Pastwatch has a slower checkout time than CVS because it must process the repository files twice. Once to create the replica snapshot and once to update the working directory. The Utah Pastwatch client has a slower checkout time than the New York Pastwatch client because it has lower bottleneck bandwidths to many of the DHT nodes.

Commit performance is comparable for Pastwatch and CVS. The difference is at most 2.5 seconds per operation. Pastwatch commits are slower than CVS because inserting data into the DHT replica is more expensive than into a single server and acquiring the lease takes additional time.

Update performance for the New York client is similar for CVS and Pastwatch. CVS update is slower at the Utah client than the New York client because the Utah client has a longer round trip time to the server and CVS uses many round trips during an update. Pastwatch updates at the Utah client are faster than at the New York client because the update operation is CPU intensive and the Utah client has a faster CPU.

6.2.3 Storage Cost:

A revtree contains every historical revision of a project; this could have resulted in a heavy storage burden, but Pastwatch stores revisions efficiently by only storing the modifications rather than entire file revisions, so the storage burden on the client replica is manageable.

After running the workload, each client database contained 7.7 megabytes of data in 4,534 blocks. 3,192 of the blocks were used to store the revtree replica. The remaining blocks were used to store the snapshots. On

disk, the BerkeleyDB database was 31 megabytes, because BerkeleyDB adds overhead for tables and a transaction log. The transaction log makes up most of the overhead but its size is bounded. In comparison, the CVS repository was 5.2 megabytes not including any replication.

The storage burden on the DHT is not very high. After running the workload described in Section 6.2.1, the resulting revtree was 4.7 megabytes in size. This means that the DHT was storing 24 megabytes of revtree data because each mutable blocks in the DHT is replicated 5 times and immutable blocks are split into 5 fragments (most of the immutable blocks are small, so each fragment is roughly the same size as the original block). Each of the 8 DHT nodes held 3 megabytes each. Again, the BerkeleyDB database adds storage overhead, so the size of the entire database on each node was 15 megabytes.

6.2.4 Many Project Members:

This section examines how Pastwatch scales with the number of project members. Pastwatch checks for new revisions at the rendezvous before most operations, so it regularly fetches each member's headpointer. This imposes an O(n) cost per project operation where n is the number of project members. This experiment uses the same setup and workload as Section 6.2.1, except the number of project members increases for each experiment. In this experiment, the New York client performs the commits and the Utah client performs an update after each commit.

Pastwatch can fetch the headpointers in parallel because it has all the headpointer addresses after retrieving the member list. Since the headpointers are small and the number of network round trips necessary to retrieve them does not depend on the number of project members, large numbers of members do not greatly affect Pastwatch operation times. Figure 12 shows that the median costs of commit and update operations increase as the number of project members increases but even at 200 members, twice as large as the most active project in Sourceforge, commits take only 1.7 seconds more and updates take 1.9 seconds more than a 2 member project. The standard deviation is between 0.4 and 0.9 seconds

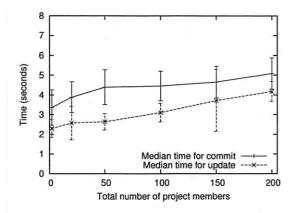


Figure 12: Median costs of commit and update in a workload for each user as the number of project members increases. Each value is the median of running the workload 10 times. The error bars are standard deviations.

and is due to varying network conditions. Ultimately, scanning the membership list is not a significant expense.

For a fixed number of new revisions, increasing the number of members who committed a revision reduces the time to retrieve the revisions because Pastwatch can retrieve revisions from different members in parallel. The worst case performance for retrieving a fixed number of new revisions occurs when a single member commits all the new revisions because Pastwatch must request them sequentially.

6.2.5 Retrieving Many Changes:

This section examines the cost of updating a user's working copy after another user has committed many new revisions. To bring a revtree up-to-date, Pastwatch needs to fetch all the new revisions which could be expensive in cases where a user has not updated her replica for some time.

These experiments use the same setup and workload as Section 6.2.1, except that only the New York client commits changes and it commits several changes before the Utah client updates its revtree and working copy. The number of commits per update varies for each experiment.

Figure 13 reports the cost of one update operation as the number of commits per update increases. The bottom curve in the figure shows only the time spent fetching headpointers. The middle curve adds the time spent fetching new revisions and delta blocks. Finally, the top curve adds in the cost of local processing to build snapshots and modify the working copy.

The top curve shows that the total cost of an update operation increases linearly with the number of revision records it needs to fetch. Decomposing the update time

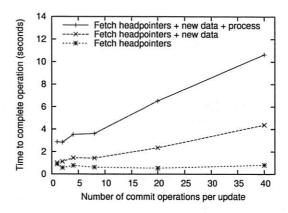


Figure 13: Median time to complete one update operation as the number of commits per update operation increases.

reveals that the linear increase is due to Pastwatch spending more time fetching revision records and delta blocks and building new snapshots. The widening gaps between the three plots illustrates that these two operations increase the runtime linearly.

7 Related Work

Version Control Systems: There are many existing VCSs but most do not attempt to support disconnected commits and ad-hoc synchronization.

Most existing VCSs are based on a client-server architecture. CVS[7], Subversion[22], Perforce[32] and Clearcase[31] all rely on a single repository server to store and manage different revisions of the project data. They do not support disconnected commits. Similarly, if the server becomes unavailable, no user can access the repository. Two users who are connected to each other cannot share changes with each other through the system when the server is unavailable.

Bitkeeper[1] uses a hierarchy of repositories to cope with server failures; it divides users into subgroups. Each subgroup commits changes to a sub-repository and propagates changes to a parent repository when they are ready to share them. A user may also have her own sub-repository, so she can read and write the repository while disconnected from the other repositories. After she reconnects, she commits locally saved changes to the parent repository. The local repository supports disconnected commits, but users in different groups cannot share changes if a parent is unavailable.

Coven[8] uses lightweight forks to support disconnected commits. When Coven users cannot contact their repository, they commit to a local lightweight fork which resembles a log. Later, when they can communicate with

the repository, they commit the lightweight fork back into the repository. Coven can support a disconnected user's commits, but directly connected users cannot share changes if the repository is unreachable.

The Monotone[4] repository resembles a revtree internally. Its repository tolerates the same kinds of inconsistencies that Pastwatch does. Monotone provides authentication by having each committer sign each revision separately whereas Pastwatch authenticates revisions with a hash tree based on a single signed reference for each writer. The hash tree makes it possible for Pastwatch to find the newest revisions when storing data in a DHT. Monotone was developed concurrently and independently from Pastwatch. In the past year, Mercurial[3] and GIT[2], have been developed based on the ideas found in Monotone and Pastwatch. We are encouraged by the use of revtree concepts in these systems.

Optimistic Replication: In addition to Pastwatch, there are many other optimistic concurrency systems that use a variety of techniques for detecting write conflicts. Using version vectors[21][24] is one common technique along with its newer variant, concise version vectors[16]. These techniques use logical clocks on each replica to impose a partial order on shared object modifications. The systems tag the shared objects with logical timestamps, which allow the systems to detect when a write-write conflict appears. Systems like Locus[30], Pangaea[28] and Ficus[20][25] use these optimistic concurrency techniques to implement optimistically replicated file systems.

Other systems, such as Bayou[29], use application specific checker functions to detect write-write conflicts. For every write, a checker verifies that a specific precondition holds before modifying the object. This ensures that the write will not damage or create a conflict with an existing object.

Coda[14][15] detects write-write conflicts by tagging each file with a unique identifier every time it is modified. When a disconnected client reconnects and synchronizes a remotely modified file, it will detect a write-write conflict because the file's tag on the server will have changed. Coda can use this technique because its file server is the ultimate authority for the file; all changes must go back to the server. Pastwatch cannot use this method because it has no central authority for its repository.

Hash histories[12] also detect write-write conflicts and resemble revtrees, but their focus is to understand how much history to maintain while still being able to detect conflicts. Pastwatch intentionally keeps all history because the version control application needs it.

All these optimistic concurrency systems provide a way to detect write-write conflicts on a shared object, but

the version control application needs more than conflict detection. It also needs the contents of all past revisions and the inheritance links between them.

It may be possible to combine version vectors with write logging to get both conflict detection and revision history, but revtrees perform both tasks simultaneously without the limitations of version vectors; revtrees do not need logical clocks and they readily support adding and removing replicas from the system.

It may also be possible to use an optimistic concurrency system to replicate an entire repository as a single shared object containing all the revision history. This approach is difficult because most existing version control systems are not designed for concurrent access and conflict resolution. The version control system's data structures must be consistent for it to function properly, but the data structures in the repository and working copies often contain interdependencies. This means the conflict resolver will need to repair the repository replicas and the working copies or else the VCS will not function properly. Although it may be possible to construct an automatic conflict resolver for an existing VCS, Pastwatch shows that a separate conflict resolver is unnecessary if the data structures are designed for concurrency. The revtree requires no active conflict resolution for its data structures and the Pastwatch working copies do not need to be repaired after concurrent writes.

8 Conclusion

We have presented Pastwatch, a distributed version control system. Under normal circumstances, Pastwatch appears like a typical client-server VCS, but Pastwatch optimistically replicates its repository on each users' computer so that each user may commit modifications while partitioned from servers and other members of the project. A user can directly synchronize his replica with other user's replicas or a rendezvous service in any pattern. All users in a given network partition can always exchange new modifications with each other.

Pastwatch supports optimistic replication and flexible synchronization between replicas because it represents the repository as a revtree data structure. Revtrees provide eventual consistency regardless of synchronization order and they detect repository level write-write conflicts using forks. Reconciling these forks is easy because they only appear at the source code level, not in the data structures of the repository and working copies.

We analyzed real-world software projects to show that disconnected commits are likely to be useful to their developers. We also showed that handling concurrent commits with forking is not a burden, even for active projects.

We implemented Pastwatch and have an initial user community with more than a year of experience using the system. Although Pastwatch is more complex than a client-server system, the implementation successfully hides those details from the users. The users have found Pastwatch to be very usable and the system has masked a number of actual failures, in contrast to the VCS it replaced.

Acknowledgments

We are grateful to the many people who contributed to this work: Butler Lampson and Barbara Liskov for their help with early revisions of the design; Frank Dabek and Emil Sit for their help with Chord/DHash; David Andersen for his help with the RON testbed; members of the PDOS research group and the anonymous reviewers that helped refine our work; and our shepherd Emin Gün Sirer.

The National Science Foundation supported this work as part of the IRIS project under Cooperative Agreement No. ANI-0225660 and with a Career grant.

References

- [1] Bitkeeper. http://www.bitkeeper.com/.
- [2] GIT. http://git.or.cz/.
- [3] Mercurial.http://www.selenic.com/mercurial/wiki/index.cgi.
- [4] Monotone. http://www.venge.net/monotone/.
- [5] Sourceforge. http://www.sourceforge.net/.
- [6] UNIX diff3 utility, 1988. http://www.gnu.org/.
- [7] B. Berliner. CVS II: Parallelizing software development. In Proc. of the USENIX Winter Conference, 1990.
- [8] Mark C. Chu-Carroll and Sara Sprenkle. Coven: Brewing better collaboration through software configuration management. In Proc. ACM SIGSOFT Conference, 2000
- [9] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Widearea cooperative storage with CFS. In *Proceedings of the ACM Symposium* on *Operating System Principles*, October 2001.
- [10] F. Dabek, J. Li, E. Sit, J. Robertson, M. Frans Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, March 2004.
- [11] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.
- [12] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), 2003.
- [13] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of the 3rd International* Workshop on Peer-to-Peer Systems, February 2004.
- [14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In Proceedings of the ACM Symposium on Operating System Principles, 1991.
- [15] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In Proc. of the USENIX Winter Conference, January 1995.

- [16] Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. In The 19th Intl. Symposium on Distributed Computing (DISC), Cracow, Poland, September 2005.
- [17] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st IPTPS*, March 2002.
- [18] D. Mazières. A toolkit for user-level file systems. In Proc. of the USENIX Technical Conference, June 2001.
- [19] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, 2(3), 1984.
- [20] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.
- [21] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, volume 9(3), 1983.
- [22] C. Michael Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. Version Control with Subversion. O'Reilly Media, Inc., 2004.
- [23] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.
- [24] D. Ratner, P. Reiher, G.J. Popek, and R. Guy. Peer replication with selective control. In *Proceedings of the First International Conference on Mobile* Data Access, 1999.
- [25] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In Proc of the USENIX Technical Conference, 1994.
- [26] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, i. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In Proceedings of ACM SIGCOMM 2005, August 2005.
- [27] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science, 2218, 2001.
- [28] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02), December 2002.
- [29] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Prin*ciples. December 1995.
- [30] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In Proceedings of the Ninth ACM Symposium on Operating Systems Principles, 1983.
- [31] Brian White. Software Configuration Management Strategies and Rational ClearCase. Addison-Wesley Professional, 2000.
- [32] Laura Wingerd. Practical Perforce. O'Reilly & Associates, 2005.
- [33] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- · encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to ;login:, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- · Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications; see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

Addison-Wesley Professional/Prentice Hall Professional • Ajava Systems, Inc. • AMD
Asian Development Bank • Cambridge Computer Services, Inc. • EAGLE Software, Inc.
Electronic Frontier Foundation • Eli Research • GroundWork Open Source Solutions
Hewlett-Packard • IBM • Intel • Interhack • The Measurement Factory • Microsoft Research
NetApp • Oracle • OSDL • Raytheon • Ripe NCC • Sendmail, Inc. • Splunk
Sun Microsystems, Inc. • Taos • Tellme Networks • UUNET Technologies, Inc.

SAGE Supporting Members

Ajava Systems, Inc. • Asian Development Bank • FOTO SEARCH Stock Footage and Stock Photography Microsoft Research • MSB Associates • Raytheon • Splunk • Taos • Tellme Networks